Funktionale und nichtfunktionale Eigenschaften

- Requirements: (nicht-)Funktionale Eigenschaften entstehen durch Erfüllung von (nicht-)funktionalen Anforderungen
- funktionale Eigenschaft: was ein Produkt tun soll
- nichtfunktionale Eigenschaft (NFE): wie ein Produkt dies tun soll
- andere Bezeichnungen NFE: Qualitäten, Quality of Service

Hardwarebasis

- Einst: Einprozessor-Systeme
- Heute: Mehrprozessor-/hochparallele Systeme
- neue Synchronisationsmechanismen erforderlich

→ unterschiedliche Hardware und deren Multiplexing

Betriebssystemarchitektur

- Einst: Monolithische und Makrokernel-Architekturen
- Heute: Mikrokernel(-basierte) Architekturen
- Exokernelbasierte Architekturen (Library-Betriebssysteme)
- Virtualisierungsarchitekturen
- Multikernel-Architekturen

→ unterschiedliche Architekturen

Ressourcenverwaltung

- Einst: Batch-Betriebssysteme, Stapelverarbeitung (FIFO)
- Heute: Echtzeitgarantien für Multimedia und Sicherheit
- echtzeitfähige Scheduler, Hauptspeicherverwaltung, Ereignismanagement, Umgang mit Überlast/Prioritätsumkehr ...

→ unterschiedliche Ressourcenverwaltung

Betriebssystemabstraktionen

- Reservierung von Ressourcen (→ eingebettete Systeme)
- Realisierung von QoS-Anforderungen (→ Multimediasysteme)
- Erhöhung der Ausfallsicherheit (→ verfügbarkeitskritisch)
- Schutz vor Angriffen und Missbrauch (→ sicherheitskritisch)
- flexiblen und modularen Anpassen des BS (→ hochadaptiv)

→ höchst diverse Abstraktionen von Hardware

Betriebssysteme als Softwareprodukte

- Betriebssystem: endliche Menge von Quellcode
- besitzen differenzierte Aufgaben → funktionale Eigenschaften
- Anforderungen an Nutzung und Pflege → Evolutionseigenschaften
- können für Betriebssysteme höchst speziell sein

→ spezielle Anforderungen an das Softwareprodukt BS

Grundlegende funktionale Eigenschaften von BS: Hardware-

Abstraktion Ablaufumgebung auf Basis der Hardware bereitstellen
Multiplexing Ablaufumgebung zeitlich/logisch getrennt einzelnen Anwendungen zuteilen
Schutz gemeinsame Ablaufumgebung gegen Fehler und Manipulation

Nichtfunktionale Eigenschaften (Auswahl) von Betriebssystemen:

- Laufzeiteigenschaften: zur Laufzeit eines Systems beobachtbar
  - Sparsamkeit und Effizienz
  - Robustheit, Verfügbarkeit
  - Sicherheit (Security)
  - Echtzeitfähigkeit, Adaptivität, Performanz
- Evolutionseigenschaften: charakterisieren (Weiter-) Entwicklung- und Betrieb eines Systems
  - Wartbarkeit, Portierbarkeit
  - Offenheit, Erweiterbarkeit

Sparsamkeit und Effizienz

Motivation

Sparsamkeit (Arbeitsdefinition): Die Eigenschaft eines Systems, seine Funktion mit minimalem Ressourcenverbrauch auszuüben → Effizienz bei Nutzung der Ressourcen
Effizienz: Der Grad, zu welchem ein System oder eine seiner Komponenten seine Funktion mit minimalem Ressourcenverbrauch ausübt. (IEEE)
Beispiele:

- mobile Geräte: Sparsamkeit mit Energie
- Sparsamkeit mit weiteren Ressourcen, z.B. Speicherplatz
- Betriebssystem (Kernel + User Space): geringer Speicherbedarf
- optimale Speicherverwaltung durch Betriebssystem zur Laufzeit
- Baugrößenoptimierung(Platinen-und Peripheriegerätegröße)
- Kostenoptimierung(kleine Caches, keine MMU, ...)
- massiv reduzierte HW-Schnittstellen (E/A-Geräte, Peripherie)

Mobile und eingebettete Systeme (kleine Auswahl)

- mobile Rechner-Endgeräte
- Weltraumfahrt und -erkundung
- Automobile
- verteilte Sensornetze (WSN)
- Chipkarten
- Multimedia-und Unterhaltungselektronik

Energieeffizienz

zeitweiliges Abschalten momentan nicht benötigter Ressourcen
Betriebssystemmechanismen

1. Dateisystem-E/A: energieeffizientes Festplatten-Prefetching
2. CPU-Scheduling: energieeffizientes Scheduling
3. Speicherverwaltung: Lokalitätsoptimierung
4. Netzwerk: energiebewusstes Routing
5. Verteiltes Rechnen: temperaturabhängige Lastverteilung

Energieeffiziente Dateizugriffe

HDD/Netzwerkgeräte/... sparen nur bei relativ langer Inaktivität Energie

- Aufgabe: kurze, intensive Zugriffsmuster → lange Inaktivität
- HDD-Geräten: Zustände mit absteigendem Energieverbrauch:
  1. Aktiv: einziger Arbeitszustand
  2. Idle: Platte rotiert, Elektronik teilweise abgeschaltet
  3. Standby: Rotation abgeschaltet
  4. Sleep: gesamte restliche Elektronik abgeschaltet
- ähnliche, noch stärker differenzierte Zustände bei DRAM
- durch geringe Verlängerungen des idle - Intervalls kann signifikant der Energieverbrauch reduziert werden

Prefetching-Mechanismus

- Prefetching („Speichervorgriff", vorausschauend) & Caching
  - Standard-Praxis bei moderner Datei-E/A
  - Voraussetzung: Vorwissen über benötigte Folge von zukünftigen Datenblockreferenzen
  - Ziel: Performanzverbesserung durch Durchsatzerhöhung und Latenzzeit-Verringerung
  - Idee: Vorziehen möglichst vieler E/A-Anforderungen an Festplatte + zeitlich gleichmäßige Verteilung verbleibender
  - Umsetzung: Caching dieser vorausschauend gelesenen Blöcke in ungenutzten PageCache

→ Inaktivität überwiegend sehr kurz → Energieeffizienz ...?

- Zugriffs-/Festplattenoperationen
  - access(x) ... greife auf Inhalt von Festplattenblock x im PageCache zu
  - fetch(x) ... hole Block x nach einem access(x) von Festplatte
  - prefetch(x) ... hole Block x ohne access(x) von Festplatte
- Fetch-on-Demand-Strategie bisher (kein vorausschauendes Lesen)
- Traditionelles Prefetching
  - traditionelle Prefetching-Strategie: bestimmt
    - wann Block von der Platte holen (HW aktiv)
    - welcher Block zu holen ist
    - welcher Block zu ersetzen ist

  1. Optimales Prefetching: Jedes *prefetch* sollte den nächsten Block im Referenzstrom in den Cache bringen, der noch nicht dort ist
  2. Optimales Ersetzen: Bei jedem ersetzenden *prefetch* sollte der Block überschrieben werden, der am spätesten in der Zukunft wieder benötigt wird
  3. „Richte keinen Schaden an": Überschreibe niemals Block A um Block B zu holen, wenn A vor B benötigt wird
  4. Erste Möglichkeit: Führe nie ein ersetzendes *prefetch* aus, wenn dieses schon vorher hätte ausgeführt werden können
- Energieeffizientes Prefetching
  - versucht Länge der Disk-Idle-Intervalle zu maximieren

  1. Optimales Prefetching: Jedes *prefetch* sollte den nächsten Block im Referenzstrom in den Cache bringen, der noch nicht dort ist
  2. Optimales Ersetzen: Bei jedem ersetzenden *prefetch* sollte der Block überschrieben werden, der am spätesten in der Zukunft wieder benötigt wird
  3. „Richte keinen Schaden an": Überschreibe niemals Block A um Block B zu holen, wenn A vor B benötigt wird
  4. Maximiere Zugriffsfolgen: Führe immer dann nach einem *fetch*/*prefetch* ein weiteres *prefetch* aus, wenn Blöcke für eine Ersetzung geeignet sind
  5. Beachte Idle-Zeiten: Unterbrich nur dann eine Inaktivitätsperiode durch ein *prefetch*, falls dieses sofort ausgeführt werden muss, um Cache-Miss zu vermeiden

Allgemeine Schlussfolgerungen

1. Hardware-Spezifikation nutzen: Modi, in denen wenig Energie verbraucht wird
2. Entwicklung von Strategien, die langen Aufenthalt in energiesparenden Modi ermöglichen und dabei Leistungsparameter in vertretbarem Umfang reduzieren
3. Implementieren dieser Strategien in Betriebssystemmechanismen zur Ressourcenverwaltung

Energieeffizientes Prozessormanagement

- CMOS z.Zt. meistgenutzte Halbleitertechnologie für Prozessor
- Komponenten für Energieverbrauch $P = P_{switch} + P_{leak} + ...$
  - P_{switch}: für Schaltvorgänge notwendige Leistung
  - P_{leak}: Verlustleistung durch verschiedene Leckströme
  - ...: weitere Einflussgrößen (technologiespezifisch)

Schaltleistung: $P_{switching}$

- Energiebedarf kapaz. Lade-/Entladevorgänge während Schaltens
- für momentane CMOS dominanter Anteil am Energieverbrauch
- Einsparpotenzial: Verringerung von Versorgungsspannung (quadratische Abhängigkeit!) und Taktfrequenz

→ längere Schaltvorgänge, größere Latenz zwischen Schaltvorgängen
⇒ Energieeinsparung nur mit Qualitätseinbußen

- Anpassung des Lastprofils (Zeit-Last? Fristen kritisch?)
- Beeinträchtigung der Nutzererfahrung (Reaktivität?)

Verlustleistung: P_{leak}

- Energiebedarf baulich bedingter Leckströme
- Hardware-Miniaturisierung → zunehmender Anteil P_{leak} an P

⇒ Leckströme kritisch für energiesparenden Hardwareentwurf

Regelspielraum: Nutzererfahrung

- Nutzererwartung: wichtigstes Kriterium zur Bewertung von auf einem Rechner aktiven Anwendungen durch Nutzer → Nutzererwartung bestimmt Nutzererfahrung
- Typ einer Anwendung entscheidet über jeweilige Nutzererwartung
  1. Hintergrund (z.B. Compiler): Gesamt-Bearbeitungsdauer, Durchsatz
  2. Echtzeit (z.B. Video-Player): ,,flüssiges" Abspielen von Video oder Musik
  3. Interaktiv (z.B. Webbrowser): Reaktivität, d.h. keine (wahrnehmbare) Verzögerung zwischen Nutzer-Aktion und Rechner-Reaktion
- Insbesondere kritisch: Echtzeit-/interaktive Anwendungen
- Reaktivität: Reaktion von Anwendungen; abhängig z.B. von
  1. **Hardware** an sich
  2. **Energieversorgung** der Hardware (z.B. Spannungspegel)
  3. **Software-Gegebenheiten** (z.B. Scheduling, Management)
- Zwischenfazit: Nutzererfahrung
  - bietet Regelspielraum für Hardwareparameter
  - Betriebssystemmechanismen zum energieeffizienten Prozessormanagement müssen mit Nutzererfahrung(jeweils erforderlicher Reaktivität) ausbalanciert werden

Energieeffizientes Scheduling

- Scheduling-Probleme beim Energiesparen: Fairness & Prioritätsumkehr
- Beispiel: Round Robin (RR) mit Prioritäten
  - E_i^{budget} ... Energiebudget von t_i
  - E_i^{limit} ... Energielimit von t_i
  - P_{limit} ... maximale Leistungsaufnahme [Energie/Zeit]
  - T ... resultierende Zeitscheibenlänge
- Problem 1: Unfaire Energieverteilung
- Problem 2: energieintensive Threads behindern nachfolgende Threads gleicher Priorität
- Problem 3: energieintensive Threads niedrigerer Priorität behindern spätere Threads höherer Priorität
- RR Strategie 1: faire Energieverteilung (einheitliche Energielimits)
  - $1 \leq i \leq 4 : E_i^{limit} = P_{limit} * T$
- faire bzw. gewichtete Aufteilung begrenzter Energie optimiert Energieeffizienz
- Problem: lange, wenig energieintensive Threads verzögern Antwort-und Wartezeiten kurzer, energieintensiver Threads
  - Lösung im Einzelfall: Wichtung per E_i^{limit}
  - globale Reaktivität → Nutzererfahrung?
- RR Strategie 2: maximale Reaktivität (→ klassisches RR)
- Problem: sparsame Threads werden bestraft durch Verfallen des ungenutzten Energiebudgets
- Idee: Ansparen von Energiebudgets → mehrfache Ausführung eines Threads innerhalb einer Scheduling-Periode
- RR Strategie 3: Reaktivität, dann faire Energieverteilung

Implementierungsfragen

- Kosten ggü. klassischem RR? (durch Prioritäten...?)
- Scheduling-Zeitpunkte?
  - welche Accounting-Operationen (Buchführung)?
  - wann Accounting-Operationen?
  - wann Verdrängung?
- Datenstrukturen?
  - ... im Scheduler → Warteschlange(n)?
  - ... im Prozessdeskriptor?
- Pro
  - Optimierung der Energieverteilung nach Schedulingzielen
  - Berücksichtigung prozessspezifischer Verbrauchsmuster
- Kontra
  - sekundäre Kosten: Energiebedarf des Schedulers, Kontextwechsel, Implementierungskosten
  - Voraussetzung: Monitoring des Energieverbrauchs
- **Alternative:** energieintensive Prozesse verlangsamen → Regelung der CPU-Leistungsparameter

Systemglobale Energieeinsparungsmaßnahmen

- Traditionelle: zu jedem Zeitpunkt Spitzen-Performanz angestrebt
  - viele Anwendungen benötigen keine Spitzen-Performanz
  - viel Hardware-Zeit in Leerlaufsituationen bzw. keine Spitzen-Performanz erforderlich
- Konsequenz (besonders für mobile Systeme)
  - Hardware mit Niedrigenergiezuständen
  - Betriebssystem kann **Energie-Management** realisieren

Hardwaretechnologien

DPM: Dynamic Power Management

- versetzt leerlaufende Hardware selektiv in Zustände mit niedrigem Energieverbrauch
- Zustandsübergänge durch Power-Manager gesteuert, bestimmte *DPM*-Strategie (Firmware) zugrunde, um gutes Verhältnis zwischen Performanz/Reaktivität und Energieeinsparung zu erzielen
- bestimmt, wann und wie lange eine Hardware in Energiesparmodus

Greedy Hardware-Komponente sofort nach Erreichen des Leerlaufs in Energiesparmodus, ,,Aufwecken" durch neue Anforderung
Time-out Energiesparmodus erst nachdem ein definiertes Intervall im Leerlauf, ,,Aufwecken" wie bei Greedy-Strategien
Vorhersage Energiesparmodus sofort nach Erreichen des Leerlaufs, wenn Heuristik vorhersagt,dass Kosten gerechtfertigt
Stochastisch Energiesparmodus auf Grundlage stochastischen Modells

DVS: Dynamic Voltage Scaling

- effizientes Verfahren zur dynamischen Regulierung von Taktfrequenz gemeinsam mit Versorgungsspannung
- Nutzung quadratischer Abhängigkeit der dynamischen Leistung von Versorgungsspannung
- Steuerung/Strategien: Softwareunterstützung notwendig
- Ziel: Unterstützung von DPM-Strategien durch Maßnahmen auf Ebene von Compiler, Betriebssystem und Applikationen
- **Betriebssystem** (prädiktives Energiemanagement)
  - kann Benutzung verschiedener Ressourcen beobachten
  - kann darüber Vorhersagen tätigen
  - kann notwendigen Performanzbereich bestimmen
- **Anwendungen** können Informationen über jeweils für sie notwendige Performanz liefern

→ Kombination mit energieeffizientem Scheduling

Speichereffizienz

- ... heißt: Auslastung des verfügbaren Speichers
- oft implizit: Hauptspeicherauslastung (memoryfootprint)
- für kleine/mobile Systeme: Hintergrundspeicherauslastung
- Maße zur Konkretisierung:
  - zeitlich: Maximum vs. Summe genutzten Speichers?
  - physischer Speicherverwaltung? → Belegungsanteil pAR
  - virtuelle Speicherverwaltung? → Belegungsanteil vAR
- Konsequenzen für Ressourcenverwaltung durch BS
  - Taskverwaltung (Accounting, Multiplexing, Fairness, ...)
  - Programmiermodell, API (dynamische Speicherreservierung)
  - Sinnfrage und Strategien virtueller Speicherverwaltung (VMM)
- Konsequenzen für Betriebssystem selbst
  - minimaler Speicherbedarf durch Kernel
  - minimale Speicherverwaltungskosten (obiger Aufgaben)

Hauptspeicherauslastung

Problem: externe Fragmentierung

- Lösungen
  - First Fit, Best Fit, WorstFit, Buddy
  - Relokation
- Kompromissloser Weg: kein Multitasking

Problem: interne Fragmentierung

- Lösung
  - Seitenrahmengröße verringern
  - Tradeoff: dichter belegte vAR → größere Datenstrukturen für Seitentabellen
- direkter Einfluss des Betriebssystems auf Hauptspeicherbelegung

  → Speicherbedarf des Kernels
  - statische (min) Größe des Kernels (Anweisungen+Daten)
  - dynamische Speicherreservierung durch Kernel
  - bei Makrokernel: Speicherbedarf von Gerätecontrollern

weitere Einflussfaktoren: Speicherverwaltungskosten

- VMM: Seitentabellengröße → Mehrstufigkeit
- Metainformationen über laufende Programme: Größe von Taskkontrollblöcken (Prozess-/Threaddeskriptoren ...)
- dynamische Speicherreservierung durch Tasks

Hintergrundspeicherauslastung

Einflussfaktoren des Betriebssystems

- statische Größe des Kernel-Images, beim Bootstrapping gelesen
- statische Größe von Programm-Images (Standards wie ELF)
- statisches vs. dynamisches Einbinden von Bibliotheken
- VMM: Größe des Auslagerungsbereichs (inkl. Teilen der Seitentabelle) für Anwendungen
- Modularisierung (zur Kompilierzeit) des Kernels: gezielte Anpassung an Einsatzdomäne möglich
- Adaptivität (zur Kompilier-und Laufzeit) des Kernels: gezielte Anpassung an sich ändernde Umgebungsbedingungen möglich

Architekturentscheidungen

- typische Einsatzgebiete sparsamer BS: eingebettete Systeme
- eingebettetes System
  - Computersystem, das in ein größeres technisches System, welches nicht zur Datenverarbeitung dient, physisch eingebunden ist
  - Wesentlicher Bestandteil dieses größeren Systems
  - Liefert Ausgaben in Form von Informationen/Daten
- spezielle, anwendungsspezifische Ausprägung der Aufgaben
  - reduzierter Umfang von HW-Abstraktion, hardwarenähere Ablaufumgebung
  - begrenzte Notwendigkeit von HW-Multiplexing & Schutz
- eng verwandte NFE: Adaptivität von sparsamen BS

- sparsame Betriebssysteme:
  - energieeffizient: geringe Architekturanforderungen an energieintensive Hardware
  - speichereffizient: Auskommen mit kleinen Datenstrukturen
- Konsequenz: geringe logische Komplexität des Betriebssystemkerns
- sekundär: Adaptivität des Betriebssystemkerns

Makrokernel (monolithischer Kernel)

- User Space:
  - Anwendungstasks
  - CPU im unprivilegierten Modus (Unix ,,Ringe" 1...3)
  - Isolation von Tasks durch Programmiermodell/VMM
- Kernel Space:
  - Kernel und Gerätecontroller (Treiber)
  - CPU im privilegierten Modus (Unix ,,Ring" 0)
  - keine Isolation
- Vergleich
  - ✓ vglw. geringe Kosten von Kernelcode (Energie, Speicher)
  - ✓ VMM nicht zwingend erforderlich
  - ✓ Multitasking nicht zwingend erforderlich
  - ✗ Kernel (inkl. Treibern) jederzeit im Speicher
  - ✗ Robustheit, Sicherheit, Adaptivität

Mikrokernel

- User Space:
  - Anwendungstasks, Kernel- und Treibertasks
  - CPU im unprivilegierten Modus
  - Isolation von Tasks durch VMM
- Kernel Space:
  - funktional minimaler Kernel (μKernel)
  - CPU im privilegierten Modus
  - keine Isolation (Kernel wird in alle vAR eingeblendet)
- Vergleich
  - ✓ Robustheit, Sicherheit, Adaptivität
  - ✓ Kernelspeicherbedarf gering, Serverprozesse nur wenn benötigt (→ Adaptivität)
  - ✗ hohe IPC-Kosten von Serverprozessen
  - ✗ Kontextwechselkosten von Serverprozessen
  - ✗ VMM, Multitasking i.d.R. erforderlich

BS: TinyOS

- Beispiel für sparsame BS im Bereich eingebetteter Systeme
- verbreitete Anwendung: verteilte Sensornetze (WSN)
- ,,TinyOS" ist ein quelloffenes, BSD-lizenziertes Betriebssystem
- für drahtlose Geräte mit geringem Stromverbrauch
- Architektur
  - monolithisch (Makrokernel) mit Besonderheiten:
  - keine klare Trennung zwischen der Implementierung von Anwendungen und BS (aber von funktionalen Aufgaben)
  - → zur Laufzeit: 1 Anwendung + Kernel
- Mechanismen:
  - kein Multithreading, keine echte Parallelität
  - → keine Synchronisation zwischen Tasks
  - → keine Kontextwechsel bei Taskwechsel
  - Multitasking realisiert durch Programmiermodell
  - nicht-präemptives FIFO-Scheduling
  - kein Paging → keine Seitentabellen, keine MMU
- in Zahlen:
  - Kernelgröße: 400 Byte
  - Kernelimagegröße: 1-4 kByte
  - Anwendungsgröße: typisch ca. 15 kB, DB: 64 kB
- Programmiermodell:
  - BS+Anwendung als Ganzes übersetzt: statische Optimierungen durch Compiler (Laufzeit, Speicherbedarf)
  - Nebenläufigkeit durch ereignisbasierte Kommunikation zw. Anwendung und Kernel
    - command: API-Aufruf, z.B. EA-Operation
    - event: Reaktion auf diesen durch Anwendung
  - sowohl commands als auch events : asynchron

BS: RIOT

- sparsames BS,optimiert für anspruchsvollere Anwendungen
- Open-Source-Mikrokernel-basiertes Betriebssystem für IoT
- Architektur
  - halbwegs: Mikrokernel
  - energiesparende Kernelfunktionalität
    - minimale Algorithmenkomplexität
    - vereinfachtes Threadkonzept → keine Kontextsicherung erforderlich
    - keine dynamische Speicherallokation
    - energiesparende Hardwarezustände vom Scheduler ausgelöst (inaktive CPU)
  - Mikrokerneldesign unterstützt komplementäre NFE: Adaptivität, Erweiterbarkeit
  - Kosten: IPC (hier gering)
- Mechanismen:
  - Multithreading-Programmiermodell
  - modulare Implementierung von Dateisystemen, Scheduler, Netzwerkstack
- in Zahlen:
  - Kernelgröße: 1,5 kByte
  - Kernelimagegröße: 5 kByte

Robustheit und Verfügbarkeit

Motivation

- allgemein: verlässlichkeitskritische Anwendungsszenarien
- Forschung in garstiger Umwelt (Weltraum)
- hochsicherheitskritische Systeme (Finanz, Cloud Dienste)
- hochverfügbare System (öffentliche Infrastruktur, Strom)
- HPC (high performance computing)

Allgemeine Begriffe

- Verlässlichkeit: Fähigkeit, eine Leistung zu erbringen, der man berechtigterweise vertrauen kann
- Untereigenschaften
  1. Verfügbarkeit (availability)
  2. Robustheit (robustness, reliability
  3. (Funktions-) Sicherheit (safety)
  4. Vertraulichkeit (confidentiality)
  5. Integrität (integrity)
  6. Wartbarkeit (maintainability) (vgl.: evolutionäre Eigenschaften)

→ nicht für alle Anwendungen sind alle Untereigenschaften erforderlich

Robustheitsbegriff

- Untereigenschaften von Verlässlichkeit: Robustheit (reliability)
- Ausfall: beobachtbare Verminderung der Leistung eines Systems, gegenüber seiner als korrekt spezifizierten Leistung
- Robustheit: Verlässlichkeit unter Anwesenheit externer Ausfälle (= Ursache außerhalb des betrachteten Systems)

Fehler, Ausfälle und ihre Vermeidung

- Fehler → fehlerhafter Zustand → Ausfall

Ausfall (failure) liegt vor, wenn tatsächliche Leistung(en), die ein System erbringt, von als korrekt spezifizierter Leistung abweichen

- Korrektheit testen/beweisen(→ formale Verifikation)

fehlerhafter Zustand (error) notwendige Ursache eines Ausfalls (nicht jeder error muss zu failure führen)

- Maskierung, Redundanz
- Isolation von Subsystemen

→ Isolationsmechanismen

Fehler (fault) Ursache für fehlerhaften Systemzustand (error), z.B. Programmierfehler

- Ausfallverhalten spezifizieren
- Ausfälle zur Laufzeit erkennen und Folgen beheben, abschwächen...

→ Micro-Reboots

Fehlerhafter Zustand

interner und externer Zustand (internal & external state)

- externer Zustand: der Teil des Gesamtzustands, der an externer Schnittstelle sichtbar wird
- interner Zustand: restlicher Teilzustand
- erbrachte Leistung: zeitliche Folge externer Zustände

Fehlerausbreitung und (externer) Ausfall

- Wirkungskette: Treiber-Programmierfehler (fault) → fehlerhafter interner Zustand des Treibers (error)
  - Ausbreitung dieses Fehlers (failure des Treibers)

  ⇒ fehlerhafter externer Zustand des Treibers

  ⇒ fehlerhafter interner Zustand des Kernels (error)

  ⇒ Kernelausfall (failure)
- Auswirkung: fehlerhafter Zustand weiterer Kernel-Subsysteme

→ Robustheit: Isolationsmechanismen

Isolationsmechanismen

- Isolationsmechanismen für robuste Betriebssysteme
  - durch strukturierte Programmierung
  - durch Adressraumisolation
- noch mehr für sichere Betriebssysteme
  - durch kryptografische Hardwareunterstützung: Enclaves
  - durch streng typisierte Sprachen und managed code
  - durch isolierte Laufzeitumgebungen: Virtualisierung

Strukturierte Programmierung

Monolithisches BS... in historischer Reinform:

- Anwendungen, Kernel, gesamte BS-Funktionalität
- programmiert als Sammlung von Prozeduren
- jede darf jede davon aufrufen, keine Modularisierung
- keine definierten internen Schnittstellen

Monolithisches Prinzip

- Ziel: Isolation zwischen Anwendungen und Betriebssystem
- Mechanismus: Prozessor-Privilegierungsebenen (user/kernelspace)
- Konsequenz: fast keine Strukturierung des Kernels

Strukturierte Makrokernarchitektur

- schwach strukturierter (monolithischer) Makrokernel

⇒ Schichtendifferenzierung (layered operating system)

- Modularisierung

Modularer Makrokernel

- Kernelfunktionen in Module unterteilt → Erweiter-/Portierbarkeit
- klar definierte Modulschnittstellen
- Module zur Kernellaufzeit dynamisch einbindbar (Adaptivität)

Fehlerausbreitung beim Makrokernel

- ✓ Wartbarkeit
- ✓ Portierbarkeit
- ✓ Erweiterbarkeit
- (begrenzt) Adaptivität
- Schutz gegen statische Programmierfehler nur durch Compiler
- ✗ kein Schutz gegen dynamische Fehler

Adressraumisolation

Private virtuelle Adressräume und Fehlerausbreitung

- private virtuelle Adressräume zweier Tasks ($i \neq j$)
- korrekte private vAR: kollisionsfreie Seitenabbildung
- Magie in Hardware: MMU (BS steuert und verwaltet...)

Robustheit: Vorteil von privaten vAR?

- ✓ nichtvertrauenswürdiger Code kann keine beliebigen physischen Adressen schreiben
- ✓ Kommunikation zwischen nvw. Code muss durch IPC-Mechanismen explizit hergestellt werden → Überwachung und Validierung zur Laufzeit möglich
- ✓ Kontrollfluss begrenzen: Funktionsaufrufe können i.A. keine AR-Grenzen überschreiten
  - → BS-Zugriffssteuerung kann nicht durch Taskfehler ausgehebelt werden
  - → unabsichtliche Terminierungsfehler(unendliche Rekursion) erschwert ...
- keine Isolation zwischen Fehlern innerhalb des Kernels

Mikrokernelarchitektur

Fortschritt ggü. Makrokernel

- Strukturierungskonzept
  - strenger durchgesetzt durch konsequente Isolation voneinander unabhängiger Kernel-Subsysteme
  - zur Laufzeit durchgesetzt → Reaktion auf fehlerhafte Zustände möglich!
- zusätzlich zu vertikaler Strukturierung des Kernels: horizontale Strukturierung eingeführt
  - → funktionale Einheiten: vertikal (Schichten)
  - → isolierte Einheiten: horizontal (private vAR)

⇒ Kernel (alle BS-Funktionalität) → μKernel (minimale BS-Funk.)

- Rest: ,,gewöhnliche" Anwendungsprozesse mit AR-isolation
- Kommunikation: botschaftenbasierte IPC (client-server OS)
- Nomenklatur: Mikrokernel und Serverprozesse

Modularer Makrokernel vs. Mikrokernel

- minimale Kernelfunktionalität:
- keine Dienste, nur allgemeine Schnittstellenfür diese
- keine Strategien, nur grundlegende Mechanismen zur Ressourcenverwaltung
- neues Problem: minimales Mikrokerneldesign

Robustheit von Mikrokernen

- = Gewinn durch Adressraumisolation innerhalb des Kernels
- ✓ kein nichtvertrauenswürdiger Code im Kernelspace, der dort beliebige physische Adressen manipulieren kann
- ✓ Kommunikation zwischen nvw. Code (nicht zur zwischen Anwendungstasks)muss durch IPC explizit hergestellt werden → Überwachung und Validierung zur Laufzeit
- ✓ Kontrollfluss begrenzen: Zugriffssteuerung auch zwischen Serverprozessen, zur Laufzeit unabhängiges Teilmanagement von Code (Kernelcode) möglich (z.B.: Nichtterminierung erkennen)
- Neu:
- ✓ nvw. BS-Code muss nicht mehr im Kernelspace laufen
- ✓ verbleibender Kernel: klein, funktional weniger komplex, leichter zu entwickeln, zu testen, evtl. formal zu verifizieren
- ✓ daneben: Adaptivität durch konsequentere Modularisierung des Kernels gesteigert

Mikrokernel: Mach

- 1975: Aleph (BS des ,,Rochester Intelligent Gateway")
- 1979/81: Accent (verteiltes BS), CMU
- Mach 3.0 (1989): einer der ersten praktisch nutzbaren μKerne
- Ziel: API-Emulation ($\neq$ Virtualisierung) von UNIX und -Derivaten auf unterschiedlichen Prozessorarchitekturen
- mehrere unterschiedliche Emulatoren gleichzeitig lauffähig
  - Emulation außerhalb des Kernels
  - Komponente im Adressraum des Applikationsprogramms
  - 1...n Server, unabhängig von Applikationsprogramm

μKernel-Funktionen

1. Prozessverwaltung
2. Speicherverwaltung
3. IPC-und E/A-Dienste, einschließlich Gerätetreiber

unterstützte Abstraktionen (→ API, Systemaufrufe):

1. Prozesse, Threads, Speicherobjekte
2. Ports (generisches, ortstransparentes Adressierungskonzept)
3. Botschaften, ... (sekundäre, von den obigen genutzte Abstraktionen)

Architektur

- Systemaufrufkosten:
  - IPC-Benchmark (1995): i486 Prozessor, 50 MHz
  - Messung mit verschiedenen Botschaftenlängen(x - Werte)
  - ohne Nutzdaten (0 Byte Botschaftenlänge): 115 μs (Tendenz unfreundlich ...)
- Bewertung aus heutiger Sicht:
  - funktional komplex
  - 153 Systemaufrufe
  - mehrere Schnittstellen, parallele Implementierungen für eine Funktion
  - → Adaptivität (Auswahl durch Programmierer)
- Fazit:
  - zukunftsweisender Ansatz
  - langsame und ineffiziente Implementierung

Lessons Learned

- Umsetzung: Designkriterien weitgehend unbekannt
- Folgen für Performanz und Programmierkomfort: [Heis19]
- ✗ ,,complex", ,,inflexible", ,,slow"
- wissen etwas über Kosten: IPC-Performanz, Kernelabstraktionen
- wissen nichts über guten μKern-Funktionsumfang und gute Schnittstellen

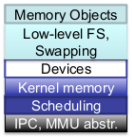
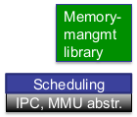
L4

Analyse des Mach-Kernels:

1. falsche Abstraktionen
2. unperformante Kernelimplementierung
3. prozessorunabhängige Implementierung

L3 und L4

- Mikrokerne der 2. Generation
- vollständige Überarbeitung des Mikrokernkonzepts

First generation Eg Mach	Second Generation Eg L4	Third generation seL4
		
180 syscalls 100 kLOC 100 μs IPC	$\sim$ 7 syscalls $\sim$ 10 kLOC $\sim$ 1 μs IPC	$\sim$ 3 syscalls 9 kLOC $0,2 - 1\mu s$ IPC

Mikrokernel - Designprinzipien

- Was gehört in einen Mikrokern?
- Konzeptsicht → Funktionalität
- Implementierungssicht → Performanz
- → 1. Generation: durch Performanzentscheidungen aufgeweicht
- → Effekt in Praxis gegenteilig: schlechte (IPC-) Performanz

Designprinzipien für Mikrokernel-Konzept

1. System interaktive und nicht vollständig vertrauenswürdige Applikationen unterstützen (→ HW-Schutz,-Multiplexing),
2. Hardware mit virtueller Speicherverwaltung und Paging

Designprinzipien

Autonomie Subsystem muss so implementiert werden, dass es von keinem anderen Subsystem gestört oder korrumpiert werden kann

Integrität Subsystem S_1 muss sich auf Garantien von S_2 verlassen können. D.h. beide Subsysteme müssen miteinander kommunizieren können, ohne dass ein drittes Subsystem diese Kommunikation stören, fälschen oder abhören kann.

L4: Speicherabstraktion

- Adressraum: Abbildung, die jede virtuelle Seite auf einen physischen Seitenrahmen abbildet oder als ,,nicht zugreifbar" markiert
- Implementierung über Seitentabellen, unterstützt durch MMU-Hardware
- Aufgabe des Mikrokernels (Schicht aller Subsysteme): muss Hardware-Konzept des Adressraums verbergen und durch eigenes Adressraum-Konzept überlagern
- Mikrokernel-Konzept des Adressraums:
  - muss Implementierung von beliebigen virtuellen Speicherverwaltungs-und -schutzkonzepten oberhalb des Mikrokernels (d.h. in den Subsystemen) erlauben
  - sollte einfach und dem Hardware-Konzept ähnlich sein
- Idee: abstrakte Speicherverwaltung
  - rekursive Konstruktion und Verwaltung der Adressräume auf Benutzer-(Server-)Ebene
  - Mikrokernel stellt dafür genau drei Operationen bereit:

grant(x) Server überträgt Seite x seines AR in AR von Empfänger
map(x) Server bildet Seite x seines AR in AR von Empfänger ab
flush(x) Server entfernt Seite x seines AR aus allen fremden AR

Hierarchische Adressräume

- Rekursive Konstruktion der Adressraumhierarchie
- Server und Anwendungenkönnen damit ihren Klienten Seiten des eigenen Adressraumes zur Verfügung stellen
- Realspeicher: Ur-Adressraum vom μKernel verwaltet
- Speicherverwaltung, Paging... außerhalb des μ-Kernels realisiert

L4: Threadabstraktion

- Thread
  - innerhalb eines Adressraumes ablaufende Aktivität
  - → Adressraumzuordnung essenziell für Threadkonzept
  - Bindung an Adressraum: dynamisch oder fest
  - Änderung einer dynamischen Zuordnung: darf nur unter vertrauenswürdiger Kontrolle erfolgen

- Designentscheidung
  - → Autonomieprinzip
  - → Konsequenz: Adressraumisolation
  - → entscheidender Grund zur Realisierung des Thread-Konzepts innerhalb des Mikrokernels

IPC

- Interprozess-Kommunikation
  - – Kommunikation über Adressraumgrenzen
  - – vertrauenswürdig kontrollierte Aufhebung der Isolation
  - → essenziell für (sinnvolles) Multitasking und -threading
- Designentscheidung
  - → Integritätsprinzip
  - → vertrauenswürdige Adressraumisolation im μKernel
  - → grundlegendes IPC-Konzepts innerhalb des Mikrokernels

Identifikatoren

- Thread-und Ressourcenbezeichner
  - – müssen vertrauenswürdig vergeben und verwaltet werden
  - → essenziell für (sinnvolles) Multitasking und -threading
  - → essenziell für vertrauenswürdige Kernel-/Server-Schnittstellen
- Designentscheidung
  - → Integritätsprinzip
  - → ID-Konzept innerhalb des Mikrokernels

Lessons Learned

1. Ein minimaler Mikrokernel
   - stellt Minimalmenge geeigneter Abstraktionen verfügbar
   - flexibel, um Implementierung beliebiger BS zu ermöglichen
   - Nutzung verschiedener Hardware-Plattformen
2. Geeignete, funktional minimale Mechanismen im μKern:
   - Adressraum mit map-, flush-, grant-Operation
   - Threadsinklusive IPC
   - eindeutige Identifikatoren
3. Wahl der geeigneten Abstraktionen: kritisch für Verifizierbarkeit, Adaptivität und optimierte Performanz des Mikrokerns
4. Bisherigen μ-Kernel-Abstraktionskonzepte: ungeeignete, zu viele, zu spezialisierte u. inflexible Abstraktionen
5. Konsequenzen für Mikrokernel-Implementierung
   - müssen für jeden Prozessortyp neu implementiert werden
   - deshalb prinzipiell nicht portierbar → L3-/L4-Prototypen: 99% Assemblercode
6. innerhalb eines Mikrokernels sind von Prozessorhardware abhängig
   (a) grundlegende Implementierungsentscheidungen
   (b) meiste Algorithmen u. Datenstrukturen
7. Fazit: Mikrokernel mit akzeptabler Performanz hardwarespezifische Implementierung minimal erforderlicher vom Prozessortyp unabhängiger Abstraktionen
8. L4 heute: Spezifikation Mikrokernels (nicht Implementierung)

Zwischenfazit

- Begrenzung von Fehlerausbreitung (→ Folgen von errors)
- konsequent modularisierte Architektur aus Subsystemen
- Isolationsmechanismen zwischen Subsystemen
- statische Isolation auf Quellcodeebene → strukturierte Programmierung
- dynamische Isolation zur Laufzeit → private virtuelle Adressräume
- Architektur, welche diese Mechanismen komponiert: Mikrokernel

✓ Adressraumisolation für sämtlichen nichtvertrauenswürdigen Code
✓ keine privilegierten Instruktionen in nvw. Code (Serverprozesse)
✓ geringe Größe (potenziell: Verifizierbarkeit) des Kernels
✓ neben Robustheit: Modularitätund Adaptivitätdes Kernels
✗ Behandlung von Ausfällen (→ abstürzende Gerätetreiber ...)

Micro-Reboots

- Kernelfehler potentiell fatal für gesamtes System
- Anwendungsfehler nicht
- → kleiner Kernel = geringeres Risiko von Systemausfällen
- → BS-Code in Serverprozessen: verbleibendes Risiko unabhängiger Teilausfälle von BS-Funktionalität
- Ergänzung zu Isolationsmechanismen notwendig
- Mechanismen zur Behandlung von Subsystem-Ausfällen
- = Mechanismen zur Behandlung Anwendungs-, Server- und Gerätetreiberfehlen
- → Micro-Reboots

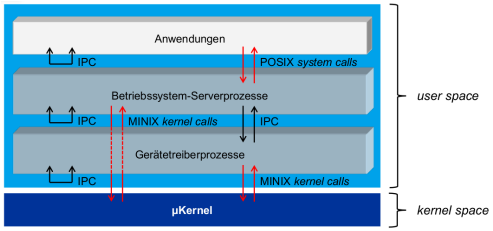
Ansatz

- kleinen (als fehlerfrei angenommenen) μKernel
- BS-Funktionalität in bedingt vertrauenswürdigen Serverprozessen
- Treiber/Anwendungen in nicht vertrauenswürdigen Prozessen
- wollen Systemausfälle verhindern durch Vermeidung von errors im Kernel → höchste Priorität
- Treiber-und Serverausfälle minimieren durch Verbergen ihrer Auswirkungen → nachgeordnete Priorität (Best-Effort-Prinzip)
- Idee: Ausfälle → Neustart durch spezialisierten Serverprozess

Beispiel-Betriebssystem: MINIX

- Ziel: robustes Betriebssystems
- → Schutz gegen Sichtbarwerden von Fehlern(= Ausfälle) für Nutzer
- Fokus auf Anwendungsdomänen: Einzelplatzrechner und eingebettete Systeme
- Anliegen: Robustheit > Verständlichkeit > geringer HW-Bedarf

Architektur



- Anwendungen (weiß): Systemaufrufe im POSIX-Standard
- Serverprozesse (grau): IPC (botschaftenbasiert), mit Kernel: spezielle MINIX-API (kernel calls), für Anwendungsprozesse gesperrt
- Betriebssystem-Serverprozesse: Dateisystem (FS), Prozessmanagement (PM), Netzwerkmanagement (Net)
- Reincarnation Server (RS) → Micro-Reboots jeglicher Serverprozesse
- Kernelprozesse: systemtask, clocktask

Reincarnation Server

- Implementierungstechnik für Micro-Reboots
- Prozesse zum Systemstart (→ Kernel Image)

system, clock Kernelprogramm
init Bootstrapping (Initialisierung rs), Fork der Login-Shell
rs Fork aller BS-Serverprozesse inkl. Gerätetreiber

Verfügbarkeit

- komplementäre NFE zu Robustheit: Verfügbarkeit (availability)
- Verbesserung von Robustheit → Verbesserung von Verfügbarkeit
- Robustheitsmaßnahmen hinreichend , nicht notwendig
- weitere komplementäre NFE: Robustheit → Sicherheit (security)
- Definition: Grad, zu welchem ein System oder eine Komponente funktionsfähig und zugänglich (erreichbar) ist, wann immer seine Nutzung erforderlich ist (IEEE)
- Anteil an Laufzeit eines Systems, in dem dieses seine spezifizierte Leistung erbringt
- $Availability = \frac{TotalUptime}{TotalLifetime} = \frac{MTTF}{MTTF+MTTR}$
- MTTR: Mean Time to Recovery, MTTF: Mean Time to Failure
- Hochverfügbarkeitsbereich (gefeierte „five nines" availability)
- Maßnahmen: Robustheit, Redundanz, Ausfallmanagement

einige Verfügbarkeitsklassen:

Verfügbarkeit	Ausfallzeit pro Jahr	Ausfallzeit pro Woche
90%	> 1 Monat	ca. 17 Stunden
99%	ca. 4 Tage	ca. 2 Stunden
99,9%	ca. 9 Stunden	ca. 10 Minuten
99,99%	ca. 1 Stunde	ca. 1 Minute
99,999%	ca. 5 Minuten	ca. 6 Sekunden
99,9999%	ca. 2 Sekunden	<< 1 Sekunde

QNX Neutrino: Hochverfügbares Echtzeit-BS

- Mikrokern-Betriebssystem
- primäres Einsatzfeld: eingebettete Systeme, z.B. Automobilbau
- Mikrokernarchitektur mit Adressraumisolation für Gerätetreiber
- (begrenzt) dynamische Micro-Rebootsmöglich
- → Maximierung der Uptime des Gesamtsystems

High-Avalability-Manager Laufzeit-Monitor der Systemdienste/Anwendungsprozesse überwacht und neustartet → μReboot-Server
High-Availability-Client-Libraries Funktionen zur transparenten automatischen Reboot für ausgefallene Server-Verbindungen

Sicherheit

Terminologie

Security IT-Sicherheit, Informationssicherheit

- Ziel: Schutz **des** Rechnersystems
- Systemsicherheit, hier besprochen

Safety Funktionale Sicherheit, Betriebssicherheit

- Ziel: Schutz **vor** einem Rechnersystem
- an dieser Stelle nicht besprochen

Sicherheitsziele

- Rechnersystem sicher gegen Schäden durch zielgerichtete Angriffe, insbesondere bzgl Informationen, die im System gespeichert, verarbeitet und übertragen werden
- für Sicherheitsziele gilt: Daten ≠ Informationen
- sukzessive Konkretisierungen bzgl anwendungsspezifischer Anforderungen

abstrakte auf konkret definierte Sicherheitsziele

Vertraulichkeit nur für einen autorisierten Nutzerkreis zugänglich
Integrität vor nicht autorisierter Veränderung geschützt
Verfügbarkeit autorisierten Nutzern in angemessener Frist zugänglich
Authentizität Urheber eindeutig erkennen
Verbindlichkeit sowohl integer als auch authentisch

Schadenspotenzial

1. Vandalismus, Terrorismus (reine Zerstörungswut)
2. Systemmissbrauch
   - illegitime Ressourcennutzung, hocheffektive Folgeangriffe
   - Manipulation von Inhalten (→ Desinformation)
3. (Wirtschafts-) Spionage und Diebstahl
   - Verlust der Kontrolle über kritisches Wissen (→ Risikotechnologien)
   - immense wirtschaftliche Schäden, z.B. Diebstahl von industriellem Know-How
4. Betrug, persönliche Bereicherung (wirtschaftliche Schäden)
5. Sabotage, Erpressung
   - Außerkraftsetzen lebenswichtiger Infrastruktur
   - Erpressung durch reversible Sabotage

Bedrohungen

1. Eindringlinge (intruders), Hacker
   - Angriff nutzt technische Schwachstelle aus (exploit)
2. Schadsoftware (malicious software, malware)
   - (teil-) automatisierte Angriffe
   - Trojanische Pferde: scheinbar nützliche Software
   - Viren, Würmer: Funktionalität zur eigenen Vervielfältigung und/oder Modifikation
   - Logische Bomben: trojanischen Pferde, deren Aktivierung an System- oder Datumsereignisse gebunden
   - Root Kits
3. Bots und Botnets
   - (weit-) verteilt ausgeführte Schadsoftware
   - eigentliches Ziel i.d.R. nicht das jeweils infizierte System

Professionelle Malware: Root Kit

- Programm-Paket, das unbemerkt Betriebssystem modifiziert, um Administratorrechte zu erlangen
- Voraussetzung: eine einzige Schwachstelle...
- ermöglichen Zugriff auf alle Funktionen und Dienste eines Betriebssystems
- Angreifer erlangt vollständige Kontrolle des Systems und kann
  - Dateien (Programme) hinzufügen bzw. ändern
  - Prozesse überwachen
  - über die Netzverbindungen senden und empfangen
  - Hintertüren für zukünftiger Angriffe platzieren
- Ziele eines Rootkits
  - seine Existenz verbergen
  - zu verbergen, welche Veränderungen vorgenommen wurden
  - vollständige und irreversible Kontrolle über BS zu erlangen
- erfolgreicher Root-Kit-Angriff ...
  - jederzeit, unentdeckbar, nicht reversibel
  - systemspezifischem Wissen über Schwachstellen
  - vollautomatisiert, also reaktiv unverhinderbar
  - uneingeschränkte Kontrolle über Zielsystem erlangen

Schwachstellen

1. Passwort (erraten, zu einfach, Brute-Force, Abfangen)
2. Programmierfehler (Speicherfehler in Anwenderprogrammen/Gerätemanagern/Betriebssystem
3. Mangelhafte Robustheit
   - keine Korrektur fehlerhafter Eingaben
   - buffer overrun/underrun (,,Heartbleed")
4. Nichttechnische Schwachstellen
   - physisch, organisatorisch, infrastrukturell
   - menschlich (→ Erpressung, socialengineering)

Zwischenfazit

- Schwachstellen sind unvermeidbar
- Bedrohungen sind unkontrollierbar
- ... und nehmen tendeziell zu!
- führt zu operationellen Risiken beim Betrieb eines IT-Systems

→ Aufgabe der BS-Sicherheit: Auswirkungen operationeller Risiken reduzieren

Sicherheitspolitiken

- Herausforderung: korrekte Durchsetzung von Sicherheitspolitiken
- Vorgehensweise: Security Engineering

Sicherheitsziele Welche Sicherheitsanforderungen muss BS erfüllen?

Sicherheitspolitik Durch welche Strategien soll es diese erfüllen?

Sicherheitsmechanismen Wie implementiert BS Sicherheitspolitik?

Sicherheitsarchitektur Wo implementiert BS S.-mechanismen?

Sicherheitspolitiken und -modelle

Kritisch für korrekten Entwurf, Spezifikation, Implementierung

- Sicherheitspolitik (Policy): Menge von Regeln, zum Erreichen eines Sicherheitsziels
- Sicherheitsmodell: formale Darstellung zur
  - Verifikation ihrer Korrektheit
  - Spezifikation ihrer Implementierung

Zugriffssteuerungspolitiken

Zugriffssteuerung (access control) Steuerung, welcher Nutzer oder Prozess mittels welcher Operationen auf welche BS-Ressourcen zugreifen darf

Zugriffssteuerungspolitik konkrete Regeln, welche die Zugriffssteuerung in einem BS beschreiben

IBAC (Identity-based AC) Politik spezifiziert, welcher Nutzer an welchen Ressourcen bestimmte Rechte hat

- Bsp.: ,,Nutzer Anna darf Brief.docx lesen"

TE (Type-Enforcement) Politik spezifiziert Rechte durch zusätzliche Abstraktion (Typen): welcher Nutzertyp an welchem Ressourcentyp bestimmte Rechte hat

- Bsp.: ,,Nutzer vom Typ Administrator darf..."

MLS (Multi-Level Security) Politik spezifiziert Rechte, indem aus Nutzern und Ressourcen hierarchische Klassen (Ebenen, ,,Levels") gleicher Kritikalität im Hinblick auf Sicherheitsziele gebildet werden

- Bsp.: ,,Nutzer der Klasse nicht vertrauenswürdig..."

DAC (Discretionary AC): Aktionen der Nutzer setzen die Sicherheitspolitik durch. Typisch: Begriff des Eigentümers von BS-Ressourcen

- Bsp.: ,,Der Eigentümer einer Datei ändert..."

MAC (Mandatory AC, obligatorische Zugriffssteuerung) Keine Beteiligung der Nutzer an der Durchsetzung einer (zentral administrierten) Sicherheitspolitik

- Bsp.: ,,Anhand des Dateisystempfads bestimmt BS..."

Traditionell: DAC, IBAC

Auszug aus der Unix-Sicherheitspolitik:

- es gibt Subjekte (Nutzer/Prozesse) und Objekte (Dateien,...)
- jedes Objekt hat einen Eigentümer
- Eigentümer legen Zugriffsrechte an Objekten fest (→ DAC)
- es gibt drei Zugriffsrechte: read, write, execute
- je Objekt gibt es drei Klassen von Subjekten, mit individuellen Zugriffsrechten: Eigentümer, Gruppe, Rest

In der Praxis

- identitätsbasierte (IBAC), wahlfreie Zugriffssteuerung (DAC)
- hohe individuelle Freiheit der Nutzer bei Durchsetzung der Politik
- hohe Verantwortung

Modellierung: Zugriffsmatrix

- Access Control Matrix (acm): Momentaufnahme der globalen Rechteverteilung zu einem definierten Zeitpunkt t
- Korrektheitskriterium: Wie kann sich dies nach t möglicherweise ändern...?
- Rechteausbreitung (privilege escalation): verursacht z.B. durch Nutzeraktion (→ DAC)
- Sicherheitseigenschaft: HRU Safety → Systemsicherheit

Modern: MAC, MLS

Sicherheitspolitik der Windows UAC (user account control)

- es gibt Subjekte (Prozesse) und Objekte (Dateisystemknoten)
- jedem Subjekt ist eine Integritätsklasse zugewiesen:

  Low nicht vertrauenswürdig
  Medium reguläre Nutzerprozesse, die Nutzerdaten manipulieren
  High Administratorprozesse, die Systemdaten manipulieren
  System (Hintergrund-) Prozesse, die ausschließlich Betriebssystemdienste auf Anwenderebene implementieren
- jedem Objekt ist analog eine dieser Integritätsklassen zugewiesen
- sämtliche DAC-Zugriffsrechte müssen mit einer Hierarchie der Integritätsklassen konsistent sein (→ MAC)
- Nutzer können Konsistenzanforderung selektiv außer Kraft setzen (→ DAC)

MAC-Modellierung: Klassenhierarchie
Beispiel Relation: $\leq =$
$\{(High, Medium), (High, Low), (Medium, Low), (High, High), (Low, Low)\}$

- repräsentiert Kritikalität hinsichtlich der Integrität
- modelliert legale Informationsflüsse zwischen Subjekten und Objekten → Schutz vor illegalem Überschreiben
- leitet Zugriffsrechte aus Informationsflüssen ab: lesen/schreiben

Modellkorrektheit: Konsistenz

- Korrektheitskriterium: Garantiert die Politik, dass acm mit $\leq$ jederzeit konsistent ist? (BLP Security)
- elevation-Mechanismus: verändert nach Nutzeranfrage (→ DAC) sowohl acm als auch $\leq$→ konsistenzerhaltend?
- anders: verändern unmittelbar nur acm → konsistenzerhaltend?

Autorisierungsmechanismen

- Sicherheitsmechanismen: Datenstrukturen und Algorithmen, welche Sicherheitseigenschaften eines BS implementieren

→ Sicherheitsmechanismen benötigt man zur Herstellung jeglicher Sicherheitseigenschaften

- Auswahl im Folgenden: Autorisierungsmechanismen und -informationen
  - Nutzerauthentisierung (Passwort-Hashing, ...)
  - Autorisierungsinformationen (Metainformationen...)
  - Autorisierungsmechanismen (Rechteprüfung, ...)
  - kryptografische Mechanismen (Hashfunktionen, ...)

Traditionell: ACLs, SUID

Autorisierungsinformationen:

- müssen Subjekte (Nutzer) bzw. Objekte (Dateien, Sockets ...) mit Rechten assoziieren → Implementierung der Zugriffsmatrix (acm), diese ist:
  - groß (→ Dateianzahl auf Fileserver)
  - dünn besetzt
  - in Größe und Inhalt dynamisch veränderlich
- Lösung: verteilte Implementierung der acm als Spaltenvektoren, deren Inhalt in den Objekt-Metadaten repräsentiert wird: Zugriffssteuerungslisten (ACLs)

ACLs: Linux-Implementierung

- objektspezifischer Spaltenvektor = Zugriffssteuerungsliste
- Dateisystem-Metainformationen: implementiert in I-Nodes

Modell einer Unix acm ...

	lesen	schreiben	ausführen
Eigentümer (u)	ja	ja	ja
Gruppe (g)	ja	nein	ja
Rest (o)	ja	nein	ja

- 3-elementige Liste, 3-elementige Rechtemenge

→ 9 Bits

- Implementierung kodiert in 16-Bit-Wort: 1 1 1 1 0 1 1 0 1

Autorisierungsmechanismen: ACL-Auswertung
Subjekte = Nutzermenge besteht aus Anzahl registrierter Nutzer

- jeder hat eindeutige UID (userID), z.B. integer- Zahl
- Dateien & Prozesse mit UID des Eigentümers versehen
  - bei Dateien: Teil des I-Nodes
  - bei Prozessen: Teil des PCB
  - standardmäßiger Eigentümer: der Ressource erzeugt hat

Nutzergruppen (groups)

- jeder Nutzer durch Eintrag in Systemdatei (/etc/group) einer/mehreren Gruppen zugeordnet (→ ACL: g Rechte)

Superuser oder root... hat grundsätzlich uneingeschränkte Rechte.

- UID = 0
- darf alle Dateien im System lesen, schreiben, ausführen
- unabhängig von ACL

ACL-Implementierung Nutzerrechte → Prozessrechte
Durchsetzung: basiert auf Prozessrechten

- Annahme: Prozesse laufen mit UID des Nutzers, der sie gestartet hat und repräsentieren Nutzerberechtigungen
- technisch: Nutzer beauftragt anderen Prozess, sich zu dublizieren (fork()) und gewünschte Programm auszuführen (exec*())
- Vererbungsprinzip

Autorisierungsmechanismen: Set-UID konsequente Rechtevererbung

- Nutzer können im Rahmen der DAC-Politik ACLs manipulieren
- Nutzer können (i.A.) jedoch keine Prozess-UIDs manipulieren
- → und genau so sollte es gem. Unix-Sicherheitspolitik auch sein!

Hintergrund

- Unix-Philosophie „ everything is a file ": BS-Ressourcen wie Sockets, E/A-Gerätehandler als Datei repräsentiert → identische Schutzmechanismen zum regulären Dateisystem
- somit: Autorisierungsmechanismen zur Begrenzung des Zugriffs auf solche Geräte nutzbar
  - *root* bzw. zweckgebundener Nutzer muss Eigentümer sein
  - ACL als `rw- --- ---` gesetzt sein
  - → Nutzerprozesse könnten z.B. nicht drucken ...
- Lösung: Mechanismus zur Rechtedelegation
  - durch weiteres Recht in ACL: SUID-Bit (setUID)
  - Programmausführung modifiziert Kindprozess, so dass UID des Programmeigentümers seine Rechte bestimmt
  - Technik: von UID abweichende Prozess-Metainformation (→ PCB) effektive UID (eUID) wird tatsächlich zur Autorisierung genutzt

Strategie für sicherheitskritische Linux-Programme

- Eigentümer *root*, SUID-Bit gesetzt
- per eUID delegiert root seine Rechte an genau solche Kindprozesse, die SUID-Programme ausführen
- → Nutzerprozesse können Systemprogramme ohne permanente root-Rechte ausführen

Weiteres Beispiel: passwd

- ermöglicht Nutzern Ändern des (eigenen) Anmeldepassworts
- Schreibzugriff auf /etc/shadow (Password-Hashes) erforderlich
- Lösung: '-rws rws r-x 1 root root 1 2005-01-20 10:00 passwd$
- passwd-Programm wird mit root-Rechten ausgeführt und passwd schreibt nur eigenen Passwort-Hash

Modern: SELinux
- 2000er: sicherheitsfokussiertes Betriebssystemprojekt für NSA
- Implementierung des μKernel-Architekturkonzepts Flask
- heute: Open Source, Teil des mainline Linux Kernels
- Klassische UNIXoide: Sicherheitspolitik fest im Kernel
- Idee SELinux: Sicherheitspolitikals eigene BS-Abstraktion
  - zentrale Datenstruktur für Regeln, die erlaubte Zugriffe auf ein SELinux-System definiert
  - erlaubt Modifikation und Anpassung an verschiedene Sicherheitsanforderungen → NFE Adaptivität ...

BS-Komponenten

- Auswertung: Security-Server, implementiert als Linux-Kernelmodul → entscheidet über alle Zugriffe auf alle Objekte
- Durchsetzung der Sicherheitspolitik: LSM Hooks
- Administration: geschrieben in Textform, muss zur Laufzeit in Security Server installiert werden

Repräsentation der Sicherheitspolitik

- physisch: in spezieller Datei, die alle Regeln enthält, die der Kernel durchsetzen muss
- Datei wird aus Menge von Quelldateien in einer Spezifikationssprache für SELinux-Sicherheitspolitiken kompiliert
- ermöglicht anforderungsspezifische SELinux-Politiken: können sich von SELinux-System zum anderen wesentlich unterscheiden
- Politik wird während des Boot-Vorgangs in Kernel geladen

Semantische Konzepte (Auswahl)

- Type Enforcement (TE): Typisierung von
  - Subjekten: Prozesse
  - Objekten der Klassen: Dateien, Sockets, Geräteschnittstellen, ...
- Rechte delegation durch Retypisierung (vgl. Unix-SUID)

Autorisierungsinformationen Security Context:
Repräsentiert SELinux-Autorisierungsinformationen für jedes Objekt (Semantik: Prozess bash läuft mit Typ `shell_t`)

Autorisierungsregeln ... werden systemweit festgelegt in dessen Sicherheitspolitik (→ MAC)
Access Vector Rules

- Autorisierungsregeln basierend auf Subjek-/Objekttypen
- Zugriffe müssen explizit gewährt werden (default-deny)
- Semantik: Erlaube ("allow") ...
  - jedem Prozess mit Typ `shell_t`
  - ausführenden Zugriff (benötigt die Berechtigung `execute`)
  - auf Dateien (also Objekte der Klassefile)
  - mit Typ `passwd_exec_t`

Autorisierungsmechanismen: passwd Revisited

- Lösung: Retypisierung bei Ausführung
- Prozess wechselt in einen aufgabenspezifischen Typ `passwd_t`
- → massiv verringertes Missbrauchspotenzial!

SELinux: weitere Politiksemantiken

- hier gezeigt: Überblick über TE
- außerdem relevant für SELinux-Politiken (und deren Administration)
  - Einschränkung von erlaubten Typtransitionen (Welches Programm darf mit welchem Typ ausgeführt werden?)
  - weitere Abstraktionsschicht: rollenbasierte Regeln (RBAC)
  - → Schutz gegen nicht vertrauenswürdige Nutzer

✓ extrem feingranulare, anwendungsspezifische Sicherheitspolitik zur Vermeidung von privilege escalation Angriffen
✓ obligatorische Durchsetzung (→ MAC, zusätzlich zu DAC)
O Softwareentwicklung: Legacy-Linux-Anwendungen ohne Einschränkung
✗ Politikentwicklung und -administration komplex
→ MAC-Mechanismen ala SELinux sind heutzutage in vielerlei Software bereits zu finden

Isolationsmechanismen
- bekannt: Isolationsmechanismen für robuste Betriebssysteme
  - strukturierte Programmierung
  - Adressraumisolation
- nun: Isolationsmechanismen für sichere Betriebssysteme
  - krypto. Hardwareunterstützung: Intel SGX Enclaves
  - sprachbasiert:
    - streng typisierte Sprachen und *managed code*: Microsoft Singularity
    - speichersichere Sprachen (Rust) + Adressraumisolation (μKernel): RedoxOS
  - isolierte Laufzeitumgebungen: Virtualisierung

Intel SGX
- SGX: Software Guard Extensions
- Ziel: Schutz von sicherheitskritischen Anwendungen durch vollständige, hardwarebasierte Isolation
- → strenggenommen kein BS-Mechanismus: Anwendungen müssen dem BS nicht mehr vertrauen
- Annahmen/Voraussetzungen

1. sämtliche Software nicht vertrauenswürdig (potenziell durch Angreifer kontrolliert)
2. Kommunikation mit dem angegriffenen System nicht vertrauenswürdig (weder vertraulich noch verbindlich)
3. kryptografische Algorithmen (Verschlüsselung und Signierung) sind vertrauenswürdig, also nicht für den Angreifer zu brechen
4. Ziel: Vertraulichkeit, Integrität und Authentizität von Anwendungen und durch sie verarbeiteten Informationen

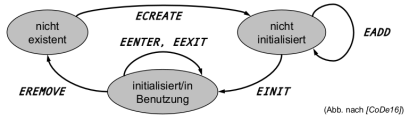
Enclaves
- Idee: geschützter Speicherbereich für Teilmenge der Seiten (Code und Daten) einer Task: Enclave Page Cache (EPC)
- Prozessor ver-und entschlüsselt EPC-Seiten

ECREATE App → Syscall → BS-Instruktion an CPU
EADD App → Syscall → BS-Instruktion an CPU

- Metainformationen für jede hinzugefügte Seite als Teil der EPC-Datenstruktur

EINIT App. → Syscall → BS-Instruktion an CPU

- finalisiert gesamten Speicherinhalt für diese Enclave
- CPU erzeugt Hashwert = eindeutige Signatur des Enclave - Speicherinhalts



(Abb. nach [CoDe16])

- Zugriff: App → CPU-Instruk. in User Mode (EENTER, EEXIT)
- CPU erfordert, dass EPC-Seiten in vAR der zugreifenden Task

SGX: Licht und Schatten

- Einführung 2015 in Skylake - Mikroarchitektur
- seither in allen Modellen verbaut, jedoch nicht immer aktiviert
- Konzept hardwarebasierter Isolation ...
- ✓ liefert erstmals die Möglichkeit zur Durchsetzung von Sicherheitspolitiken auf Anwendungsebene
- O setzt Vertrauen in korrekte (und nicht böswillige) Hardwarevoraus
- O Dokumentation und Entwicklerunterstützung (im Ausbau ...)
- ✗ schützt durch Enclaves einzelne Anwendungen aber nicht System
- ✗ steckt in praktischer Eigenschaften (Performanz, Speicher) noch in den Anfängen

Sicherheitsarchitekturen

- Voraussetzung zum Verstehen jeder Sicherheitsarchitektur
  - Verstehen des Referenzmonitorprinzips
  - frühe Forschungen durch US-Verteidigungsministerium
  - Schlüsselveröffentlichung: Anderson-Report (1972)
  - → fundamentalen Eigenschaften zur Charakterisierung von Sicherheitsarchitekturen
- Begriffe des Referenzmonitorprinzips kennen wir schon
  - Abgrenzung passiver Ressourcen (Objekte, z.B. Dateien)
  - von Subjekten (aktiven Elementen, Prozess) durch BS

Referenzmonitorprinzip

→ sämtliche Autorisierungsentscheidungen durch zentralen Mechanismus = Referenzmonitor

- Bewertet jeden Zugriffsversuch eines Subjekts auf Objekt durch Anwendung einer Sicherheitspolitik (security policy)
- Architekturbeschreibung, wie Zugriffe auf Ressourcen, die Sicherheitspolitik erlaubt, eingeschränkt werden
- Autorisierungsentscheidungen: basieren auf sicherheitsrelevanten Eigenschaften jedes Subjekts und jedes Objekts

Referenzmonitor ist eine Architekturkomponenten, die

RM 1 bei sämtlichen Subjekt/Objekt-Interaktionen involviert sind → Unumgehbarkeit (total mediation)
RM 2 geschützt sind vor unautorisierter Manipulation → Manipulationssicherheit (tamperproofness)
RM 3 hinreichend klein und wohlstrukturiert sind, für formale Analysemethoden → Verifizierbarkeit (verifyability)

Referenzmonitor in Betriebssystemen

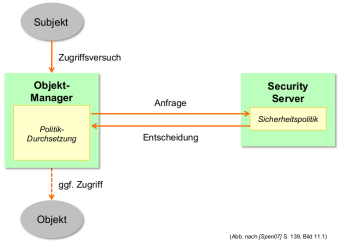
Nahezu alle Betriebssysteme implementieren irgendeine Form eines Referenzmonitors

- Subjekte, Objekte
- Regeln einer Sicherheitspolitik charakterisiert
- Unumgehbarkeit, Manipulationssicherheit
- Verifizierbarkeit ihrer Sicherheitsarchitektur

Beispiel: Standard-Linux

- Subjekte (Prozesse) → haben reale Nutzer-Identifikatoren (UIDs)
- Objekte (Dateien) → haben ACLs (,,rwxrw—-")
- Regeln der Sicherheitspolitik → hart codiert, starr
- Sicherheitsattribute, → Objekten zugeordnet, modifizierbar

Man beurteile die Politikimplementierung in dieser Architektur bzgl. Unumgehbarkeit, Manipulationssicherheit und Verifizierbarkeit
Referenzmonitorimplementierung: Flask



SELinux-Architektur: Security Server

- Security Server: Laufzeitumgebung für Politik in Schutzdomäne des Kerns
- Objektmanager: implementiert in allen BS-Diensten mittels ,,Linux Security Module Framework "
- Objektmanager zur Verwaltung verschiedener Objektklassen
- spiegeln Diversität und Komplexität von Linux BS-Abtraktionen wider: Dateisysteme, Netzwerk, IPC, ...
- jedes Subsystem von SELinux zuständig für
  1. Erzeugung neuer Objekte
  2. Zugriff auf existierende Objekte
- Beispiele: Prozess-Verwaltung, Dateisystem, Networking-System

Dateisystem als Objektmanager

- Durch Analyse von Linux - Dateisystem und zugehöriger API wurden zu überwachenden Objektklassen identifiziert
- ergibt sich unmittelbar aus Linux-API: Dateien, Verzeichnisse, Pipes
- feingranularere Objektklassen für durch Dateien repräsentierte Objekte (Unix: ,,everything is a file")

Permissions (Zugriffsrechte)

- für jede Objektklasse: Menge an Permissions definiert, um Zugriffe auf Objekte dieser Klasse zu kontrollieren
- Permissions: abgeleitet aus Dienstleistungen, die Linux-Dateisystem anbietet
- → Objektklassen gruppieren verschiedene Arten von Zugriffsoperationen auf verschiende Arten von Objekten
- z.B. Permissions für alle ,,Datei"-Objektklassen (Auswahl ...)

Trusted Computing Base (TCB)

Begriff zur Bewertung von Referenzmonitorarchitekturen

= notwendige Hard-und Softwarefunktionen eines IT-Systems um alle Sicherheitsregeln durchzusetzen

- besteht üblicherweise aus
  1. Laufzeitumgebung der Hardware (nicht E/A-Geräte)
  2. verschiedenen Komponenten des Betriebssystem-Kernels
  3. Benutzerprogrammen mit sicherheitsrelevanten Rechten
- Betriebssystemfunktionen, die Teil der TCB sein müssen, beinhalten Teile des Prozess-, Speicher-, Datei-, E/A-Managements

Echtzeitfähigkeit

Jedes System, bei dem der Zeitpunkt, zu dem der Output erzeugt wird, von Bedeutung ist. Dies liegt in der Regel daran, dass die Eingabe einer Bewegung in der physischen Welt entspricht und die Ausgabe sich auf dieselbe Bewegung beziehen muss. Die Verzögerung zwischen Eingabe- und Ausgabezeit muss für eine akzeptable Aktualität ausreichend klein sein.
Spektrum von Echtzeitsystemen

1. Regelungssysteme: z.B. eingebettete Systeme, Flugsteuerung
2. Endanwender-Rechnersysteme: z.B. Multimediasysteme
3. Lebewesen: Menschen, Tiere, z.B. Gesundheitsüberwachung

- Murphy's Law: If something can go wrong, it will got wrong
- Murphy's Constant: Damage to an object is proportional to its value
- Johnson's Law: If a system stops working, it will do it at the worst possible time
- Sodd's Law: Sooner or later, the worst possible combination of circumstances will happen
- Realisierung von Echtzeiteigenschaften: komplex und fragil

Antwortzeit Zeitintervall, das ein System braucht, um (irgend)eine Ausgabe als Reaktion auf (irgend)eine Eingabe zu erzeugen
Frist
- bei EZS ist genau dieses Δt kritisch, d.h. je nach Art des Systems darf dieses auf keinen Fall zu groß werden
- Frist (deadline) d, die angibt bis zu welchem Zeitpunkt spätestmöglich die Reaktion erfolgt sein muss, bzw. wie groß das Intervall Δt maximal sein darf

Echtzeitfähigkeit und Korrektheit
- Wird genau dieses maximale Zeitintervall in die Spezifikation eines Systems einbezogen, bedeutet dies, dass ein Echtzeitsystem nur dann korrekt arbeitet, wenn seine Reaktion bis zur spezifizierten Frist erfolgt
- Frist trennt korrektes von inkorrektem Verhalten des Systems

Harte und weiche Echtzeitsysteme
- Praktische Anwendungen erfordern oft Unterscheidung
- hartes EZS: keine Frist jemals überschreiten
- weiches EZS: maßvolles Frist Überschreiten tolerierbar

Charakteristika von Echtzeit-Prozessen

- reale Echtzeitanwendungen beinhalten periodische oder aperiodische Prozesse (oder Mischung aus beiden)
- Periodische Prozesse
  - zeitgesteuert (typisch: periodische Sensorauswertung)
  - oft: kritische Aktivitäten → harte Fristen
- Aperiodische Prozesse
  - ereignisgesteuert
  - Abhängig von Anwendung: harte oder weiche Fristen

Periodische Prozesse (pP)

häufigster Fall bei Echtzeit-Anwendungen
Prozessaktivierung ereignisgesteuert oder zeitgesteuert
Prozesse, die Eingangsdaten verarbeiten: meist ereignisgesteuert
Prozesse, die Ausgangsdaten erzeugen: meist zeitgesteuert
Fristen
- hart oder weich (anwendungsabhängig)
- innerhalb einer Anwendung sind sowohl Prozesse mit harten oder weichen Fristen möglich
- Frist: spätestens am Ende der aktuellen Periode, möglich auch frühere Frist

Modellierung unendliche Folge identischer Aktivierungen: Instanzen, aktiviert mit konstanter Rate (Periode)
Aufgaben des Betriebssystems
- Wenn alle Spezifikationen eingehalten werden, muss Betriebssystem garantieren, dass
- zeitgesteuerte pP: mit ihrer spezifizierten Rate aktiviert werden und ihre Frist einhalten können
- ereignisgesteuerte pP: ihre Frist einhalten können

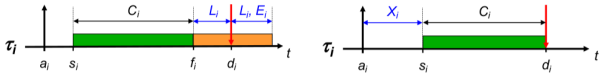
Aperiodische Prozesse (aP)

typisch für unregelmäßig auftretende Ereignisse, z.B.:

- Überfahren der Spurgrenzen, Unterschreiten des Sicherheitsabstands → Reaktion des Fahrassistenzsystems
- Nutzereingaben in Multimediasystemen (→ Spielkonsole)

Prozessaktivierung ereignisgesteuert
Fristen oft weich (aber anwendungsabhängig)
Aufgaben des Betriebssystems unter Einhaltung der Prozessspezifikationen muss BS für Einhaltung der Fristen sorgen
Modellierung bestehen ebenfalls aus (maximal unendlicher) Folge identischer Aktivierungen (Instanzen); aber: Aktivierungszeitpunkte nicht regelmäßig (möglich: nur genau eine Aktivierung)

Parameter von Echtzeit-Prozessen



Ankunftszeitpunkt a_i Prozess wird ablauffähig
Startzeitpunkt s_i Prozess beginnt mit Ausführung
Beendigungszeitpunkt f_i Prozess beendet Ausführung
Frist (deadline) d_i Prozess sollte Ausführung spätestens beenden
Bearbeitungszeit (computation time) C_i Zeit die Prozessor zur Bearbeitung der Instanz benötigt (ohne Unterbrechungen)
Unpünktlichkeit (lateness) $L_i = f_i - d_i$ Zeit um die Prozess früher/später als Frist beendet
Verspätung (exceeding time) $E_i = max(0, L_i)$ Zeitbetrag, den ein Prozess noch nach seiner Frist aktiv ist
Spielraum (Laxity) $X_i = d_i - a_i - C_i$ maximale Verzögerungszeit bis Frist beendet werden kann ($f_i = d_i$)
criticality Konsequenzen einer Fristüberschreitung (hart/weich)
Wert V_i Ausdruck relativer Wichtigkeit eines Prozesses

Echtzeitfähige Betriebssysteme

1. Algorithmen, die Rechnersysteme echtzeitfähig machen
   - grundlegende Algorithmen zum Echtzeitscheduling
   - Besonderheiten der Interruptbehandlung
   - Besonderheiten der Speicherverwaltung
2. Probleme, die behandelt werden müssen
   - Prioritätsumkehr
   - Überlast
   - Kommunikation-und Synchronisationsprobleme

Echtzeitscheduling

Scheduling wichtigster Einflussfaktor auf Zeitverhalten des Gesamtsystems
Echtzeit-Scheduling unter Berücksichtigung der Fristen

Fundamentale/wichtigste Strategien

1. Ratenmonotones Scheduling (RM)
2. Earliest Deadline First (EDF)

Annahmen der Scheduling-Strategien

1. Alle Instanzen eines periodischen Prozesses t_i treten regelmäßig und mit konstanter Rate auf. Das Zeitintervall T_i zwischen zwei aufeinanderfolgenden Aktivierungen heißt Periode des Prozesses
2. Alle Instanzen eines periodischen Prozesses t_i haben den gleichen Worst-Case-Rechenzeitbedarf C_i
3. Alle Instanzen eines periodischen Prozesses t_i haben die gleiche relative Frist D_i, welche gleich der Periodendauer T_i ist
4. Alle Prozessesind kausal unabhängig voneinander (d.h. keine Vorrang- und Betriebsmittel-Restriktionen)
5. Kein Prozess kann sich selbst suspendieren, z.B. E/A-Op
6. Alle Prozesse werden mit ihrer Aktivierung sofort rechenbereit
7. Jeglicher Betriebssystem-Overhead wird vernachlässigt

5-7 sind weitere Annahmen des Scheduling Modells

Ratenmonotones Scheduling (RM)

- Voraussetzung: nur periodische Prozesse/Threads
- Strategie RM
  - Prozess/Thread mit höchster Ankunftsrate bekommt höchste statische Priorität
  - Kriterium: Wie oft pro Zeiteinheit wird Prozess bereit?
  - Scheduling-Zeitpunkt: nur einmal zu Beginn bzw. wenn neuer periodischer Prozess auftritt
  - präemptiv: keine Verdrängung gleicher Prioritäten
- Optimalität: Unter allen Verfahren mit festen Prioritäten optimaler Algorithmus
- Prozessor-Auslastungsfaktor
  - Bei Menge von n Prozessen $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$
  - mit $\frac{C_i}{T_i}$ Anteil an Prozessorzeit für jeden Prozess t_i
  - und Zeit U zur Ausführung der gesamten Prozessmenge
- Prozessorlast: U ist folglich Maß für die durch Prozessmenge verursachte Last am Prozessor → Auslastungsfaktor
- Planbarkeitsanalyse einer Prozessmenge
  - allgemein kann RM Prozessor nicht 100% auslasten
  - kleinste obere Grenze des Auslastungsfaktors U_{lub}
  - lub: ,,least upper bound"
- Obere Auslastungsgrenze bei RM
  - nach Buttazzo bei n Prozessen: $U_{lub} = n(2^{\frac{1}{n}} - 1)$
  - für $n \to \infty$ konvergiert U_{lub} zu $ln\ 2 \approx 0,6931...$
  - Wert nicht überschritten → beliebige Prozessmengen

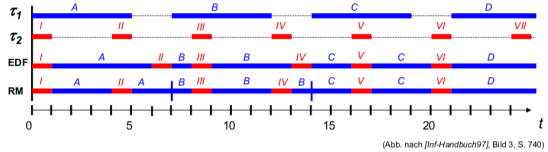
Earliest Deadline First (EDF)

- Voraussetzung: kann periodische/aperiodische Prozesse planen
- Optimalität: EDF in Klasse der Schedulingverfahren mit dynamischen Prioritäten: optimaler Algorithmus
- Strategie EDF
  - Prozess mit frühester Frist höchste dynamische Priorität
  - Scheduling-Zeitpunkt: Bereitwerden eines Prozesses
  - präemptiv: keine Verdrängung gleicher Prioritäten
- Planbarkeitsanalyse
  - mit Regeln $1 - 7$ max. Prozessorauslastung: $U_{lub} = 1 \rightarrow$ Auslastung bis 100%
  - Menge von n Tasks planbar: $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

  ← U>1 übersteigt die verfügbare Prozessorzeit; folglich kann niemals eine Prozessmenge mit dieser Gesamtauslastung planbar sein

  → Beweis durch Widerspruch. Annahme: $U \leq 1$ und die Prozessmenge ist nicht planbar. Dies führt zu einem Schedule mit Fristverletzung zu einem Zeitpunkt t_2

Vergleich: EDF vs. RM



(Abb. nach [Inf-Handbuch97], Bild 3, S. 740)

- RM
  - Prozessorwechsel: 16
  - im allgemeinen Fall nicht immer korrekte Schedules bei 100% Auslastung
  - statisch Implementiert: jeweils eine Warteschlange pro Priorität
  - Einfügen und Entfernen von Tasks: $O(1)$
- EDF
  - Prozessorwechsel: 12
  - erzeugt auch bei Prozessorauslastung bis 100% (immer) korrekte Schedules
  - dynamisch Implementiert: balancierter Binärbaum zur Sortierung nach Prioritäten
  - Einfügen und Entfernen von Tasks: $O(log\ n)$

Prozesstypen in Multimedia-Anwendungen

1. Echte periodische Multimedia-Prozesse (weiche Fristen)
   (a) pünktliche periodische Prozesse mit konstantem Prozessorzeitbedarf C für jede Instanz (unkomprimierte Audio- und Videodaten)
   (b) pünktliche periodische Prozesse mit unterschiedlichem C einzelner Instanzen (komprimierte Audio- und Videodaten)
   (c) unpünktliche periodische Prozesse: verspätet/verfrühte
2. Prozesse nebenläufiger Nicht-Multimedia-Anwendungen
   - interaktiv: keine Fristen , keine zu langen Antwortzeiten Ansatz, maximal tolerierbare Verzögerung
   - Hintergrund: zeitunkritisch, keine Fristen, dürfen nicht verhungern

RC Algorithmus

- Ziel: spezifikationstreue Prozesse nicht bestrafen durch Fristüberschreitung aufgrund abweichender Prozesse
- Idee
  - grundsätzlich: Scheduling nach frühester Frist aufsteigend

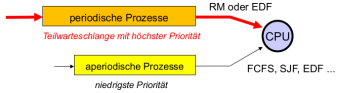
  → für vollständig spezifikationstreue Prozessmenge wie EDF
  - Frist einer Instanz wird dynamisch angepasst: basierend auf derjenigen Periode, in der sie eigentlich sein sollte
  - Bsp.: $U_i = \frac{20}{40} = \frac{1}{2}$ (spez. Aktivitätsrate $0,5$/Periode)
- Variablen
  - a_i: Ankunftszeit der zuletzt bereitgewordenen Instanz
  - t_i^{virt}: virtuelle verbrauchte Zeit in aktueller Periode
  - c_i^{virt}: verbrauchte Netto-Rechenzeit
  - d_i: dynamische Frist von t_i für Priorität (EDF)
- Strategie
  - für eine bereite (lauffähige) Instanz von t_i: adaptiere dynamisch d_i basierend auf t_i^{virt}
  - für eine bereit gewordene Instanz von t_i: aktualisiere t_i^{virt} auf akt. Systemzeit (t) → etwaiger ,,Zeitkredit" verfällt
- Zeitpunkte, zu denen der Scheduler aktiv wird
  - aktuell laufender Prozess t_i blockiert: $RC(t_i)$
  - Prozesse $t_i...t_j$ werden bereit: $for\ x \in [i, j] : RC(t_x)$
  - periodischer ,,clock tick" (Scheduling Interrupt) $RC(t_i)$

Umgang mit abweichenden Prozessen unter RC

Auswirkung auf verschiedene Prozesstypen

pünktlich Einhaltung der Frist in jeder Periode garantiert
verspätet nur aktuelle Periode betrachtet, Nachholen ,,ausgelassener Perioden" nicht möglich
gierig Prozessorentzug, sobald andere lauffähige Prozesse frühere Fristen aufweisen
nicht-periodische Hintergrundprozesse pro ,,Periode" wird spezifizierte Prozessorrate garantiert

Umgang mit gemischten Prozessmengen



- rechenbereite Prozesse auf 2 Warteschlangen aufgeteilt (einfache Variante eines Mehr-Ebenen-Scheduling)
- Warteschlange 1
  - alle periodischen Prozesse
  - mit höchster Priorität mittels RM oder EDF bedient
- Warteschlange 2
  - alle aperiodischen Prozesse
  - nur bedient, wenn keine wartenden Prozesse in W1

Hintergrund-Scheduling: Vor- und Nachteile

- Hauptvorteil einfache Implementierung
- Nachteile
  - Antwortzeit **aperiodischer Prozesse** kann zu lang werden
  - → Verhungern möglich
  - nur für relativ zeitunkritische aperiodische Prozesse

Optimierung: Server-Prozess

- Prinzip: periodisch aktivierter Prozess benutzt zur Ausführung aperiodischer Prozessoranforderungen
- Beschreibung Server-Prozess: durch Parameter äquivalent zu wirklichem periodischen Prozess
- Arbeitsweise Server-Prozess folgend
- geplant mit gleichem S-Algorithmus wie periodische Prozesse
- zum Aktivierungszeitpunkt vorliegende aperiodische Anforderungen bedient bis zur Kapazität des Servers
- keine aperiodischen Anforderungen: Server suspendiert sich bis Beginn der nächsten Periode
- Kapazität in jeder Server-Periode neu "aufgeladen"

Optimierung: Slack-Stealing

- Prinzip: passiver Prozess ,,slack stealer" (kein periodischer Server)
- so viel Zeit wie möglich für aperiodische Anforderungen sammeln
- realisiert durch ,,slackstealing" bei periodischen Prozessen
- letztere auf Zeit-Achse so weit nach hinten geschoben, dass Frist und Beendigungszeitpunkt zusammenfallen
- Sinnvoll, da Beenden vor Frist keine Vorteile bringt

→ Verbesserung der Antwortzeiten für aperiodische Anforderungen

Prioritätsumkehr

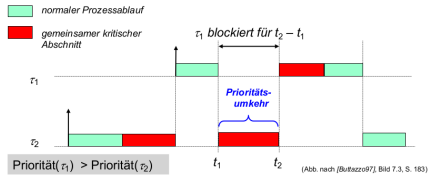
Mechanismen zur Synchronisation und Koordination sind häufige Ursachen für kausale Abhängigkeiten zwischen Prozessen

- *kritischer Abschnitt*: Sperrmechanismen stellen wechselseitigen Ausschluss durch nebenläufige Prozesse sicher
- Benutzung von exklusiven, nichtentziehbaren Betriebsmitteln

→ Wenn ein Prozess einen kritischen Abschnitt betreten hat, darf er aus diesem nicht verdrängt werden

- Konflikt: kritische Abschnitte vs. Echtzeit-Prioritäten
- Prozess mit höherer Priorität ablauffähig → muss abwarten bis niederpriorisierter Prozess kritischen Abschnitt verlässt
- (zeitweise) Prioritätsumkehr möglich

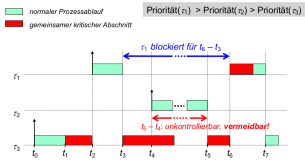
Ursache der Prioritätsumkehr



- Prioritätsumkehr bei Blockierung an nichtentziehbarem, exklusivem Betriebsmittel
- → unvermeidlich

Folgen der Prioritätsumkehr

- Kritisch bei zusätzlichen Prozessen mittlerer Priorität



- Lösung: Priority Inheritance Protocol (PIP)

Überlast

- Definition: kritische Situation, bei der benötigte Menge an Prozessorzeit die Kapazität des vorhandenen Prozessors übersteigt

→ nicht alle Prozesse können Fristen einhalten

- Hauptrisiko: kritische Prozesse können Fristen nicht einhalten → Gefährdung funktionaler und anderer nichtfkt. Eigenschaften (→ harte Fristen!)
- Stichwort: ,,graceful degradation" statt unkontrollierbarer Situation → Wahrung von Determinismus

Wichtigkeit eines Prozesses

- Unterscheidung zwischen Zeitbeschränkungen (Fristen) und tatsächlicher Wichtigkeit eines Prozesses für System
- Wichtigkeit eines Prozesses ist unabhängig von seiner Periodendauer und irgendwelchen Fristen
- z.B. kann Prozess trotz späterer Frist wichtiger als anderer mit früherer Frist sein

Umgang mit Überlast: alltägliche Analogien

1. Weglassen weniger wichtiger Aktionen (kein Frühstück...)
2. Verkürzen von Aktivitäten (Katzenwäsche...)
3. Kombinieren (kein Frühstück + Katzenwäsche + ungekämmt)

Wichtung von Prozessen

- Parameter V für jeden Prozess/Thread einer Anwendung
- spezifiziert relative Wichtigkeit eines Prozesses/Thread im Verhältnis zu anderen der gleichen Anwendung
- bei Scheduling: V stellt zusätzliche Randbedingung (primär: Priorität aufgrund von Frist, sekundär: Wichtigkeit)

Obligatorischer und optionaler Prozessanteil

- Aufteilung der Gesamtberechnung eines Prozesses in zwei Phasen
- Möglichkeit der Nutzung des anpassbaren Prozessorzeitbedarfs
- Bearbeitungszeitbedarf eines Prozesses zerlegt in
  1. obligatorischer Teil: unbedingt und immer ausführen → liefert bedingt akzeptables Ergebnis
  2. optionaler Teil: nur bei ausreichender Lapazität ausführen → verbessert erzieltes Ergebnis
- Prinzip in unterschiedlicher Weise verfeinerbar

Echtzeit-Interruptbehandlung

- Fristüberschreitung durch ungeeignete Interruptbearbeitung
- Interrupt wird nur registriert (deterministischer Zeitaufwand)
- tatsächliche Bearbeitung der Interruptroutine muss durch Scheduler eingeplant werden → Pop-up Thread

Echtzeit-Speicherverwaltung

- Hauptanliegen: Fristen einhalten
- unkontrollierbare Verzögerungen der Prozessbearbeitung vermeiden
- Ressourcenzuordnung, deswegen:
  1. keine Ressourcen-Zuordnung ,,on-demand" sondern ,,Pre-Allokation" (=Vorab)
  2. keine dynamische Ressourcenzuordnung, sondern Zuordnung maximal benötigter Menge bei Pre-Allokation

Hauptspeicherverwaltung

- bei Anwendung existierender Paging-Systeme
- durch unkontrolliertes Ein-/Auslagern ,,zeitkritischer" Seiten (-inhalte): unkontrollierbare Zeitverzögerungen möglich
- Technik: ,,Festnageln" von Seiten im Speicher (Memory Locking)

Sekundärspeicherverwaltung

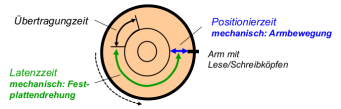
- Primärziel: Wahrung der Echtzeitgarantien
  - naheliegend: EA-Scheduling nach Fristen → EDF
  - für Zugriffsreihenfolge auf Datenblöcke: lediglich deren Fristen maßgebend (weitere Regeln existieren nicht)
- Resultat bei HDDs
  - ineffiziente Bewegungen der Lese-/Schreibköpfe
  - nichtdeterministische Positionierzeiten
  - geringer Durchsatz
- Fazit: Echtzeit-Festplattenscheduling → Kompromiss zwischen Zeitbeschränkungen und Effizienz
- bekannte Lösungen: Modifikation/Kombination von EDF

→ realisierte Strategien:

1. SCAN-EDF (Kopfbewegung in eine Richtung bis Mitte-/Randzylinder; EDF über alle angefragten Blöcke in dieser Richtung)
2. Group Sweeping (SCAN nach Fristen gruppenweiser Bedienung)
3. Mischstrategien

- Vereinfachung: o.g. Algorithmen i.d.R. zylinderorientiert

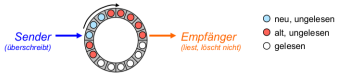
→ berücksichtigen bei Optimierung nur Positionierzeiten (Positionierzeit meist >> Latenzzeit)



Kommunikation und Synchronisation

- zeitlichen Nichtdeterminismus vermeiden: Interprozess-Kommunikation
  - Minimierung blockierender Kommunikationsoperationen
  - indirekte Kommunikation → Geschwindigkeitsausgleich
  - keine FIFO-Ordnungen (nach Fristen priorisieren)
- Synchronisation: keine FIFO-Ordnungen

Cyclic Asynchronous Buffer (CAB)



Kommunikation zwischen 1 Sender und n Empfängern

- nach erstem Schreibzugriff: garantiert niemals undefinierte Wartezeiten durch Blockierung von Sender/Empfänger
- Lesen/Überschreiben in zyklischer Reihenfolge:
- Most-Recently-Written (MRW) Zeiger auf jüngstes, durch Sender vollständig geschriebenes Element
- Least-Recently-Written (LRW) Zeiger auf ältestes durch Sender geschriebenes Element
- sowohl *MRW* als auch *LRW* können ausschließlich durch Sender manipuliert werden → keine inkonsistenten Zeiger durch konkurrierende Schreibzugriffe
- sowohl *MRW* als auch *LRW* zeigen niemals auf ein Element, das gerade geschrieben wird → keine inkonsistenten Inhalte durch konkurrierende Schreib-/Lesezugriffe
- Regeln für Sender
  - muss **nach** jedem Schreiben *MRW* auf geschriebenes Element setzen
  - muss **bevor** *LRW* geschrieben wird *LRW* inkrementieren

- Regel für Empfänger: muss immer nach Lesen von *MRW* als nächstes *LRW* anstelle des Listennachbarn lesen

Sonderfall 1: Empfänger schneller als Sender

- nach Zugriff auf MRW muss auf Lesesequenz bei LRW fortgesetzt werden → transparenter Umgang mit nicht-vollem Puffer
- Abschwächung der Ordnungsgarantien:Empfänger weiß nur, dass Aktualität der Daten zwischen LRW und MRW liegt
- Empfänger niemals durch leeren Puffer blockiert

Sonderfall 2: Sender schneller als Empfänger

- Schreiben in Puffer in Reihenfolge der Elemente → keine blockierenden Puffergrenzen → niemals Blockierung des Senders
- keine Vollständigkeitsgarantien: Empfänger kann nicht sicher sein, eine temporal stetige Sequenz zu lesen

→ Szenarien, in denen Empfänger sowieso nur an aktuellsten Daten interessiert (z.B. Sensorwerte)

Konkurrierende Zugriffe

- ... sind durch Empfänger immer unschädlich (da lesend)
- ... müssen vom Sender nach Inkrementieren von LRW nicht-blockierend erkannt werden (Semaphormodell ungeeignet)
- schnellerer Sender überspringtein gesperrtes Element durch erneutes Inkrementieren von LRW, MRW muss nachziehen

Architekturen

- müssen Echtzeitmechanismen unterstützen; ermöglicht entsprechende Strategien zur Entwicklungs-oder Laufzeit
- müssen funktional geringe Komplexität aufweisen → theoretische und praktische Beherrschung von Nichtdeterminismus
- Architekturen für komplementäre NFE
  - Sparsamkeit → hardwarespezifische Kernelimplementierung
  - Adaptivität → μKernel, Exokernel
- zu vermeiden
  - starke HW-Abstraktion → Virtualisierungsarchitekturen
  - Kommunikation und Synchronisationskosten → verteilte BS
  - Hardwareunabhängigkeit und Portabilität → vgl. Mach

Beispiel-Betriebssysteme

VRTX (Versatile Real-Time Executive)

- Entwickler: Hunter & Ready
- Eckdaten: Makrokernel
- war erstes kommerzielles Echtzeitbetriebssystem für eingebettete Systeme
- Nachfolger (1993 bis heute): Nucleus RTOS (Siemens)
- Anwendung: Eingebettete Systeme in Automobilen, Mobiltelefone
- Einsatzgebiete im Hubble-Weltraumteleskop

VxWorks

- Entwickler: Wind River Systems (USA)
- Eckdaten: modularer Makrokernel
- Erfolgsfaktor: POSIX-konforme API
- ähnlich QNX: ,,skalierbarer" Kernel, zuschneidbar auf Anwendungsdomäne (→ Adaptivitätsansatz)
- Anwendung: eingebettete Systeme, Luft-und Raumfahrt, Unterhaltungselektronik
- Einsatzgebiete: NASA Mars Rover, SpaceX Dragon

DRYOS

- Entwickler: Canon Inc.
- Eckdaten: Mikrokernel(Größe: 16 kB)
- Echtzeit-Middleware (Gerätetreiber → Objektive)
- Anwendungen: AE-und AF-Steuerung/-Automatik, GUI, Bildbearbeitung, RAW-Konverter, ...
- POSIX-kompatible Prozessverwaltung

DROPS (Dresden Real-Time Operating System)

- Entwickler: TU Dresden, Lehrstuhl Betriebssysteme
- Eckdaten: Multi-Server-Architektur auf Basis eines L4-Mikrokerns

Adaptivität

Motivation

- als unmittelbar geforderte NFE:
  - eingebettete Systeme
  - Systeme in garstiger Umwelt (Meeresgrund, Arktis, Weltraum, ...)
  - Unterstützung von Cloud-Computing-Anwendungen
  - Unterstützung von Legacy-Anwendungen
- Beobachtung: genau diese Anwendungsdomänen fordern typischerweise auch andere wesentliche NFE (s.bisherige Vorlesung ...)
- → Adaptivität als komplementäre NFE zur Förderung von
  - Robustheit: funktionale Adaptivitätdes BS reduziert Kernelkomplexität (→ kleiner, nicht adaptiver μKernel)
  - Sicherheit: wie Robustheit:TCB-Größe → Verifizierbarkeit, außerdem: adaptive Reaktion auf Bedrohungen
  - Echtzeitfähigkeit: adaptive Scheduling-Strategie (vgl. RC), adapt. Überlastbehandlung, adapt. Interruptbehandlungs-und Pinning-Strategien
  - Performanz: Last-und Hardwareadaptivität
  - Erweiterbarkeit: adaptive BS liefern oft hinreichende Voraussetzungen der einfachen Erweiterbarkeit von Abstraktionen, Schnittstellen, Hardware-Multiplexing-und -Schutzmechanismen (Flexibility)
  - Wartbarkeit: Anpassung des BS an Anwendungen, nicht umgekehrt
  - Sparsamkeit: Lastadaptivitätvon CPUs, adaptive Auswahl von Datenstrukturen und Kodierungsverfahren

Adaptivitätsbegriff

- Adaptability: ,,see Flexibility. " [Marciniak94]
- Flexibility:
  - ,,The ease with which a system or a component can be modified for use in applications or environments other than those for which it was specifically designed." (IEEE)
  - für uns: entspricht Erweiterbarkeit
- Adaptivität: (unsere Arbeitsdefinition)
  - Die Fähigkeit eines Systems, sich an ein breites Spektrum verschiedener Anforderungen anpassen zu lassen.
  - = ... so gebaut zu sein, dass ein breites Spektrum verschiedener nicht funktionaler Eigenschaften unterstützt wird.
  - letztere: komplementär zur allgemeinen NFE Adaptivität

Roadmap

- in diesem Kapitel: gleichzeitig Mechanismen und Architekturkonzepte
- Adaptivität jeweils anhand komplementärer Eigenschaften dargestellt:
  - Exokernel: { Adaptivität } ∪ { Performanz, Echtzeitfähigkeit,Wartbarkeit, Sparsamkeit }
  - Virtualisierung: { Adaptivität } ∪ { Wartbarkeit, Sicherheit, Robustheit }
  - Container: { Adaptivität } ∪ { Wartbarkeit, Portabilität, Sparsamkeit }
- Beispielsysteme:
  - Exokernel-Betriebssysteme: Aegis/ExOS, Nemesis, MirageOS
  - Virtualisierung: Vmware, VirtualBox, Xen
  - Containersoftware: Docker

Exokernelarchitektur

- Grundfunktion von Betriebssystemen
  - physische Hardware darstellen als abstrahierte Hardware mit komfortableren Schnittstellen
  - Schnittstelle zu Anwendungen (API) : bietet dabei exakt die gleichen Abstraktionen der Hardware für alle Anwendungen an, z.B.
    - **Prozesse:** gleiches Zustandsmodell, gleiches Threadmodell
    - **Dateien:** gleiche Namensraumabstraktion
    - **Adressräume:** gleiche Speicherverwaltung (VMM, Seitengröße, Paging)
    - **Interprozesskommunikation:** gleiche Mechanismen für alle Anwendungsprozesse
- Problem:
  - Implementierungsspielraumfür Anwendungen wird begrenzt:

1. Vorteile domänenspezifischer Optimierungender Hardwarebenutzung können nicht ausgeschöpft werden → **Performanz, Sparsamkeit**
2. die Implementierung existierender Abstraktionen kann bei veränderten Anforderungen nicht an Anwendungen angepasst werden → **Wartbarkeit**
3. Hardwarespezifikationen, insbesondere des Zeitverhaltens (E/A, Netzwerk etc.), werden von Effekten des BS-Management überlagert → **Echtzeitfähigkeit**

- Idee von Exokernel-Architekturen:

Exokernelmechanismen

- Designprinzip von Exokernelmechanismen:
  - Trennung von Schutz und Abstraktion der Ressourcen
  - Ressourcen-Schutz und -Multiplexing: verbleibt beim Betriebssystemkernel(dem Exokernel)
  - Ressourcen-Abstraktion (und deren Management): zentrale Aufgabe der Library-Betriebssysteme
    - → autonome Management-Strategien durch in Anwendungen importierte Funktionalität
  - Resultat:
    1. systemweit(durch jeweiliges BS vorgegebene) starre Hardware-Abstraktionen vermieden
    2. anwendungsdomänenspezifische Abstraktionen sehr einfach realisierbar
    3. (Wieder-) Verwendung eigener und fremder Managementfunktionalität wesentlich erleichtert → komplementäre NFEn! (Performanz, EZ-Fähigkeit, Sparsamkeit, ...)
- Funktion des Exokernels:
  - Prinzip: definiert Low-level-Schnittstelle
    - ,,low-level" = so hardwarenah wie möglich, bspw. die logische Schnittstelle eines elektronischen Schaltkreises/ICs (→ Gerätetreiber ⊆ Library-BS!)
    - Bsp.: der Exokernelmuss den Hauptspeicher schützen, aber nicht verstehen, wie dieser verwaltet wird → Adressierung ermöglichen ohne Informationen über Seiten, Segmente, Paging-Attribute, ...
  - Library-Betriebssysteme: implementieren darauf jeweils geeignete anwendungsnahe Abstraktionen
    - Bsp.: Adressraumsemantik, Seitentabellenlayout und -verwaltung, Paging-und Locking-Verfahren, ...
  - Anwendungsprogrammierer: wählen geeignete Library-Betriebssysteme bzw. schreiben ihre eigenen Exokernelmechanismen
- prinzipielle Exokernelmechanismen am Beispiel Aegis/ExOS [Engler+95]
  - Der Exokernel...

    * *implementiert:* Multiplexing der Hardware-Ressourcen
    * *exportiert:* geschützte Hardware-Ressourcen

- minimal: drei Arten von Mechanismen
    1. Secure Binding: erlaubt geschützte Verwendung von Hardware-Ressourcen durch Anwendungen, Behandlung von Ereignissen
    2. Visible ResourceRevocation: beteiligt Anwendungen am Entzug von Ressourcen mittels (kooperativen) Ressourcen-Entzugsprotokolls
    3. Abort-Protokoll: erlaubt ExokernelBeendigung von Ressourcenzuordnungen bei unkooperativen Applikationen

Secure Binding

- Schutzmechanismus, der Autorisierung (→ Library-BS)zur Benutzung einer Ressource von tatsächlicher Benutzung (→ Exokernel) trennt
- implementiert für den Exokernelerforderliches Zuordnungswissenvon (HW-)Ressource zu Mangement-Code (der im Library-BS implementiert ist)
- → "Binding" in Aegis implementiert als Unix-Hardlinkauf Metadatenstruktur zu einem Gerät im Kernelspeicher (,,remember: everythingisa file...")
- Zur Implementierung benötigt:
    - Hardware-Unterstützung zur effizienten Rechteprüfung (insbes. HW-Caching)
    - Software-Caching von Autorisierungsentscheidungen im Kernel (bei Nutzung durch verschiedene Library-BS)
    - Downloadingvon Applikationscode in Kernel zur effizienten Durchsetzung (quasi: User-Space-Implementierung von Systemaufrufcode)
- einfach ausgedrückt: ,,Secure Binding" erlaubt einem ExokernelSchutz von Ressourcen, ohne deren Semantik verstehen zu müssen.

Visible Resource Revocation

- monolithische Betriebssysteme: entziehen Ressourcen ,,unsichtbar" (invisible), d.h. transparent für Anwendungen
    - Vorteil: im allgemeinen geringere Latenzzeiten, einfacheres und komfortableres Programmiermodell
    - Nachteil: Anwendungen(hier: die eingebetteten Library-BS) erhalten keine Kenntnis über Entzug,bspw. aufgrund von Ressourcenknappheit etc.

  → erforderliches Wissen für Management-Strategien!
- Exokernel-Betriebssysteme: entziehen(überwiegend) Ressourcen ,,sichtbar" → Dialog zwischen Exokernel und Library-BS
    - Vorteil: effizientes Management durch Library-BS möglich (z.B. Prozessor: nur tatsächlich benötigte Register werden bei Entzug gespeichert)
    - Nachteil : Performanz bei sehr häufigem Entzug, Verwaltungs-und Fehlerbehandlungsstrategien zwischen verschiedenen Library-BS müssen korrekt und untereinander kompatibelsein...

  → Abort - Protokoll notwendig, falls dies nicht gegeben ist

Abort - Protokoll

- Ressourcenentzug bei unkooperativen Library-Betriebssystemen (Konflikt mit Anforderung durch andere Anwendung/deren Library-BS: Verweigerung der Rückgabe, zu späte Rückgabe, ...)
- notwendig aufgrund von Visible Ressource Revocation
- Dialog:
    - Exokernel: ,,Bitte Seitenrahmen x freigeben."
    - Library-BS: ,,..."
    - Exokernel: ,,Seitenrahmen x innerhalb von 50 μs freigeben!"
    - Library-BS: ,,..."
    - Exokernel: (führt Abort-Protokoll aus)
    - Library-BS: X (,,Abort" in diesem Bsp. = Anwendungsprozess terminieren)
- In der Praxis:
    - harte Echtzeit-Fristen (,, innerhalb von 50 μs") in den wenigsten Anwendungen berücksichtigt

      → Abort = lediglich Widerruf aller Secure Bindings der jeweiligen Ressource für die unkooperativeAnwendung, nicht deren Terminierung (= unsichtbarerRessourcenentzug)

      → anschließend: Informieren des entsprechenden Library-BS
    - ermöglicht sinnvolle Reaktion des Library-BS (in Library-BS wird ,,Repossession"-Exceptionausgelöst, so dass auf Entzug geeignet reagiert werden kann)
    - bei zustandsbehafteten Ressourcen (→ CPU): Exokernelkann diesen Zustand auf Hintergrundspeicher sichern → Management-Informationen zum Aufräumen durch Library-BS

Exokernelperformanz

- Was macht Exokern-Architekturen adaptiv(er)?
    - Abstraktionen und Mechanismen des Betriebssystems können den Erfordernissen der Anwendungen angepasst werden
    - (erwünschtes) Ergebnis: beträchtliche Performanzsteigerungen (vgl. komplementäre Ziel-NFE: Performanz, Echtzeitfähigkeit, Wartbarkeit, Sparsamkeit)

Performanzstudien

1. Aegis mit Library-BS ExOS (MIT: Dawson Engler, Frans Kaashoek)
2. Xok mit Library-BS ExOS (MIT)
3. Nemesis (Pegasus-Projekt, EU)
4. XOmB (U Pittsburgh)
5. ...

Aegis/ExOSals erweiterte Machbarkeitsstudie [Engler+95]

1. machbar: sehr effiziente Exokerne
    - Grundlage: begrenzte Anzahl einfacher Systemaufrufe (Größenordnung 10) und Kernel-interne Primitiven (,,Pseudo-Maschinenanweisungen"), die enthalten sein müssen
2. machbar: sicheres Hardware-Multiplexing auf niedriger Abstraktionsebene (,,low-level") mit geringem Overhead
3. traditionelle Abstraktionen (VMM, IPC) auf Anwendungsebene effizient implementierbar → einfache Erweiterbarkeit, Spezialisierbarkeitbzw. Ersetzbarkeit dieser Abstraktionen
4. für Anwendungen: hochspezialisierte Implementierungen von Abstraktionen generierbar, die genau auf Funktionalität und Performanz-Anforderungen dieser Anwendung zugeschnitten
5. geschützte Kontrollflussübergabe: als IPC-Primitive im Aegis-Kernel, 7-mal schnellerals damals beste Implementierung (vgl. [Liedtke95], Kap. 3)
6. Ausnahmebehandlung bei Aegis: 5-mal schneller als bei damals bester Implementierung
7. durch Aegis möglich: Flexibilität von ExOS, die mit Mikrokernel-Systemen nicht erreichbar ist:
    - Bsp. VMM: auf Anwendungsebene implementiert, wo diese sehr einfach mit DSM-Systemen u. Garbage-Kollektoren verknüpfbar
8. Aegis erlaubt Anwendungen Konstruktion effizienter IPC-Primitiven ($\Delta\mu$Kernel: nicht vertrauenswürdige Anwendungen können keinerlei spezialisierte IPC-Primitiven nutzen, geschweige denn selbst implementieren)

Xok/ExOS

- praktische Weiterentwicklung von Aegis: Xok
- für x86-Hardware implementiert
- Kernel-Aufgaben (wie gehabt): Multiplexing von Festplatte, Speicher, Netzwerkschnittstellen, ...
- Standard Library-BS (wie bei Aegis): ExOS
    - ,,Unix as a Library"
    - Plattform für unmodifizierte Unix-Anwendungen (csh, perl, gcc, telnet, ftp, ...)
- z.B. Library-BS zum Dateisystem-Management: C-FFS
    - hochperformant (im Vergleich mit Makrokernel-Dateisystem-Management)
    - Abstraktionen und Operationen auf Exokernel-Basis (u.a.): Inodes, Verzeichnisse, physische Dateirelokation(→ zusammenhängendes Lesen)
    - Secure Bindings für Metadaten-Modifikation
- Forschungsziele:
    - Aegis: Proof-of-Concept
    - XOK: Proof-of-Feasibility (Performanz)

Zwischenfazit: Exokernelarchitektur

- Ziele:
    - Performanz, Sparsamkeit: bei genauer Kenntnis der Hardware ermöglicht deren direkte BenutzungAnwendungsentwicklern Effizienzoptimierung
    - Wartbarkeit: Hardwareabstraktionen sollen flexibel an Anwendungsdomänen anpassbar sein, ohne das BS modifizieren/wechseln zu müssen
    - Echtzeitfähigkeit: Zeitverhaltendes Gesamtsystems durch direkte Steuerung der Hardware weitestgehend durch (Echtzeit-) Anwendungen kontrollierbar
- Idee:
    - User-Space:anwendungsspezifische Hardwareabstraktionen im User-Space implementiert
    - Kernel-Space:nur Multiplexing und Schutz der HW-Schnittstellen
    - in der Praxis: kooperativer Ressourcenentzug zwischen Kernel, Lib. OS
- Ergebnisse:
    - hochperformanteHardwarebenutzung durch spezialisierte Anwendungen
    - funktional kleiner Exokernel(→ Sparsamkeit, Korrektheit des Kernelcodes)
    - flexible Nutzung problemgerechterHW-Abstraktionen (readymade Lib. OS)
    - keine Isolation von Anwendungen (→ Parallelisierbarkeit: teuer und mit schwachen Garantien; → Robustheit und Sicherheit der Anwendungen: nicht umsetzbar)

Virtualisierung

- Ziele (zur Erinnerung):
    - Adaptivität
    - Wartbarkeit, Sicherheit, Robustheit

  → auf gleicher Hardware mehrere unterschiedliche Betriebssysteme ausführbar machen
- Idee:

Ziele von Virtualisierung

- Adaptivität: (ähnlich wie bei Exokernen)
    - können viele unterschiedliche Betriebssysteme - mit jeweils unterschiedlichen Eigenschaften ausgeführt werden damit können: Gruppen von Anwendungen auf ähnliche Weise jeweils unterschiedliche Abstraktionen etc. zur Verfügung gestellt werden

- Wartbarkeit:
  - Anwendungen - die sonst nicht gemeinsam auf gleicher Maschine lauffähig - auf einer phyischenMaschine ausführbar
  - ökonomische Vorteile: Cloud-Computing, Wartbarkeit von Legacy-Anwendungen
- Sicherheit:
  - Isolation von Anwendungs-und Kernelcode durch getrennte Adressräume (wie z.B. bei Mikrokern-Architekturen)
  - somit möglich:
    1. Einschränkung der Fehlerausbreitung → angreifbare Schwachstellen
    2. Überwachung der Kommunikation zwischen Teilsystemen
  - darüber hinaus: Sandboxing (vollständig von logischer Ablaufumgebung isolierte Software, typischerweise Anwendungen → siehe z.B. Cloud-Computing)
- Robustheit:
  - siehe Sicherheit!

Architekturvarianten - drei unterschiedliche Prinzipien:

1. Typ-1 - Hypervisor (früher: VMM - ,,Virtual MachineMonitor")
2. Typ-2 - Hypervisor
3. Paravirtualisierung

Typ-1 - Hypervisor

- Idee des Typ- 1 - Hypervisors:
  - Kategorien traditioneller funktionaler Eigenschaften von BS:
    1. Multiplexing & Schutz der Hardware (ermöglicht Multiprozess-Betrieb)
    2. abstrahierte Maschine** mit ,,angenehmerer" Schnittstelle als die reine Hardware (z.B. Dateien, Sockets, Prozesse, ...)
- Typ- 1 - Hypervisor trennt beide Kategorien:
  - läuft wie ein Betriebssystem unmittelbar über der Hardware
  - bewirkt Multiplexing der Hardware, liefert aber keine erweiterte Maschine** an Anwendungsschicht → ,,Multi-Betriebssystem-Betrieb"
- Bietet mehrmals die unmittelbare Hardware-Schnittstelle an, wobei jede Instanz eine virtuelle Maschine jeweils mit den unveränderten Hardware-Eigenschaften darstellt (Kernel u. User Mode, Ein-/Ausgaben usw.).
- Ursprünge: Time-Sharing an Großrechnern
  - Standard-BS auf IBM-Großrechner System/360: OS/360
  - reines Stapelverarbeitungs-Betriebssystem (1960er Jahre)
  - Nutzer (insbes. Entwickler) strebten interaktive Arbeitsweise an eigenem Terminal an → timesharing (MIT, 1962: CTSS)
    - IBM zog nach: CP/CMS, später VM/370 → z/VM
    - CP: Control Program → Typ- 1 - Hypervisor
    - CMS: ConversationalMonitor System → Gast-BS
  - CP lief auf ,,blanker" Hardware (Begriff geprägt: ,,bare metal hypervisor")
    - lieferte Menge virtueller Kopiender System/360-Hardware an eigentliches Timesharing-System
    - je eines solche Kopie pro Nutzer → unterschiedliche BS lauffähig (da jede virtuelle Maschine exakte Kopie der Hardware)
    - in der Praxis: sehr leichtgewichtiges, schnelles Einzelnutzer-BS als Gast → CMS (heute wäre das wenig mehr als ein Terminal-Emulator...)
- heute: Forderungen nach Virtualisierung von Betriebssystemen
  - seit 1980er: universeller Einsatz des PC für Einzelplatz- und Serveranwendungen → veränderte Anforderungen an Virtualisierung
  - Wartbarkeit: vor allem ökonomische Gründe:
    1. Anwendungsentwicklung und -bereitstellung: verschiedene Anwendungen in Unternehmen, bisher auf verschiedenen Rechnern mit mehreren (oft verschiedenen) BS, auf einem Rechner entwickeln und betreiben (Lizenzkosten!)
    2. Administration: einfache Sicherung, Migration virtueller Maschinen
    3. Legacy-Software
  - später: Sicherheit, Robustheit → Cloud-Computing-Anwendungen
- ideal hierfür: Typ- 1 - Hypervisor
  - ✓ Gast-BS angenehm wartbar
  - ✓ Softwarekosten beherrschbar
  - ✓ Anwendungen isolierbar

Hardware-Voraussetzungen

- Voraussetzungen zum Einsatz von Typ-1-HV
  - Ziel: Nutzung von Virtualisierung auf PC-Hardware
  - systematische Untersuchung der Virtualisierbarkeit von Prozessoren bereits 1974 durch Popek & Goldberg [Popek&Goldberg74]
  - Ergebnis:
    - Gast-BS (welches aus Sicht der CPU im User Mode - also unprivilegiert läuft) muss sicher sein können, dass privilegierte Instruktionen (Maschinencode im Kernel) ausgeführt werden
    - dies geht nur, wenn tatsächlich der HV diese Instruktionen ausführt!
    - dies geht nur, wenn CPU bei jeder solchen Instruktion im Nutzermodus Kontextwechsel zum HV ausführen, welcher Instruktion emuliert!
- virtualisierbare Prozessoren bis ca. 2006:
  - ✓ IBM-Architekturen(bekannt: PowerPC, bis 2006 Apple-Standard)
  - ✗ Intel x86-Architekturen (386, Pentium, teilweise Core i)

Privilegierte Instruktionen **ohne** Hypervisor

- kennen wir schon: Instruktion für Systemaufrufe

1. User Mode: Anwendung bereitet Befehl und Parameter vor
2. User Mode: Privilegierte Instruktion (syscall/Trap - Interrupt) → CPU veranlasst Kontext-und Privilegierungswechsel, Ziel: BS-Kernel
3. Kernel Mode: BS-Dispatcher (Einsprungpunkt für Kernel-Kontrollfluss) behandelt Befehl und Parameter, ruft weitere privilegierte Instruktionen auf (z.B. EA-Code)

Privilegierte Instruktionen mit Typ- 1 - Hypervisor(1)

- zum Vergleich: Instruktion für Systemaufrufe des Gast-BS

1. User Mode: Anwendung bereitet Befehl und Parameter vor
2. User Mode: Trap → Kontext-und Privilegierungswechsel, Ziel: Typ-1-HV
3. Kernel Mode: HV-Dispatcher ruft Dispatcher im Gast-BS auf
4. User Mode: BS-Dispatcher behandelt Befehl und Parameter, ruft weitere privilegierte Instruktionenauf (z.B. EA-Code) → Kontext-und Privilegierungswechsel, Ziel: Typ-1-HV
5. Kernel Mode: HV führt privilegierte Instruktionen anstelle des Gast-BS aus

Sensible und privilegierte Instruktionen: Beobachtungen an verschiedenen Maschinenbefehlssätzen: [Popek&Goldberg74]

- ∃ Menge an Maschinenbefehlen, die nur im Kernel Mode ausgeführt werden dürfen (Befehle zur Realisierung von E/A, Manipulation der MMU, ...)

  → sensible Instruktionen
- ∃ Menge an Maschinenbefehlen, die Wechsel des Privilegierungsmodus auslösen (x86: Trap), wenn sie im User Mode ausgeführt werden

  → privilegierte Instruktionen
- Prozessor ist virtualisierbarfalls (notw. Bed.): sensible Instruktionen ⊆ privilegierte Instruktionen
- Folge: jeder Maschinenbefehl, der im Nutzermodus nicht erlaubt ist, muss einen Privilegierungswechsel auslösen (z.B. Trap generieren)
- kritische Instruktionen = sensible Instruktionen \ privilegierte Instruktionen
  - Befehle, welche diese Bedingung verletzen → Existenz im Befehlssatz führt zu nicht-virtualisierbarem Prozessor
- Beispiele für sensible Instruktionen bei Intel x86:
  - hlt: Befehlsabarbeitung bis zum nächsten Interrupt stoppen
  - invlpg: TLB-Eintrag für Seite invalidieren
  - lidt: IDT (interrupt descriptor table) neu laden
  - mov auf Steuerregistern
  - ...
- Beispiel: Privilegierte Prozessorinstruktionen
  - Bsp.: write - Systemaufruf
  - Anwendungsprogramm schreibt String in Puffer eines Ausgabegeräts ohne Nutzung der libc Standard-Bibliothek: `asm ( int $0x80" ); /* interrupt 80 (trap) */`
  - Interrupt-Instruktion veranlasst Prozessor zum Kontextwechsel: Kernelcode im privilegierten Modus ausführen

Vergleich: Privilegierte vs. sensible Instruktionen
Folgen für Virtualisierung

- privilegierte Instruktionen bei virtualisierbaren Prozessoren
- bei Ausführung einer privilegierten Instruktion in virtueller Maschine: immer Kontrollflussübergabe an im Kernel-Modus laufende Systemsoftware - hier Typ-1-HV
- HV kann (anhand des virtuellen Privilegierungsmodus) feststellen:
  1. ob sensible Anweisung durch Gast-BS
  2. oder durch Nutzerprogramm (Systemaufruf!) ausgelöst
- Folgen:
  1. privilegierte Instruktionen des Gast-Betriebssystems werden ausgeführt → ,,trap-and-emulate"
  2. Einsprung in Betriebssystem, hier also Einsprung in Gast-Betriebssystem → Upcall durch HV
- privilegierte Instruktionen bei nicht virtualisierbaren Prozessoren
  - solche Instruktionen typischerweise ignoriert!

Intel-Architektur ab 386

- dominant im PC-und Universalrechnersegment ab 1980er
- keine Unterstützung für Virtualisierung ...
- kritische Instruktionen im User Mode werden von CPU ignoriert
- außerdem: in Pentium-Familie konnte Kernel-Code explizit feststellen, ob er im Kernel- oder Nutzermodus läuft → Gast-BS trifft (implementierungsabhängig) evtl. fatal fehlerhafte Entscheidungen
- Diese Architekturprobleme (bekannt seit 1974) wurden 20 Jahre lang im Sinne von Rückwärtskompatibilität auf Nachfolgeprozessoren übertragen ...

  - erste virtualisierungsfähige Intel-Prozessorenfamilie (s. [Adams2006]): VT, VT-x® (2005)
  - dito für AMD: SVM, AMD-V® (auch 2005)

Forschungsarbeit 1990er Jahre

- verschiedene akademische Projekte zur Virtualisierung bisher nicht virtualisierbarer Prozessoren
- erstes und vermutlich bekanntestes: DISCO- Projekt der University of Stanford
- Resultat: letztlich VMware (heute kommerziell) und Typ-2-Hypervisors...

Typ-2-Hypervisor

Virtualisierung ohne Hardwareunterstützung:

- keine Möglichkeit, trap-and-emulate zu nutzen
- keine Möglichkeit, um
  1. korrekt (bei sensiblen Instruktionen im Gast-Kernel) den Privilegierungsmodus zu wechseln
  2. den korrekten Code im HV auszuführen

Übersetzungsstrategie in Software:

- vollständige Übersetzung des Maschinencodes, der in VM ausgeführt wird, in Maschinencode, der im HV ausgeführt wird
- praktische Forderung: HV sollte selbst abstrahierte HW-Schnittstelle zur Ausführung des (komplexen!) Übersetzungscodes zur Verfügung haben (z.B. Nutzung von Gerätetreibern)
- → Typ-2-HV als Kompromiss:
  - korrekte Ausführung von virtualisierter Software auf virtualisierter HW
  - beherrschbare Komplexität der Implementierung

aus Nutzersicht

- läuft als gewöhnlicher Nutzer-Prozess auf Host-Betriebssystem (z.B. Windows oder Linux)
- VMware bedienbarwie physischer Rechner (bspw. erwartet Bootmedium in virtueller Repräsentation eines physischen Laufwerks)
- persistente Daten des Gast-BS auf virtuellem Speichermedium (tatsächlich: Image-Datei aus Sicht des Host-Betriebssystems)

Mechanismus: Code-Inspektion

- Bei Ausführung eines Binärprogramms in der virtuellen Maschine (egal ob Bootloader, Gast-BS-Kernel, Anwendungsprogramm): zunächst inspiziert Typ-2-HV den Code nach Basisblöcken
  - Basisblock: Befehlsfolge, die mit privilegierten Befehlen oder solchen Befehlen abgeschlossen ist, die den Kontrollfluss ändern (sichtbar an Manipulation des Programm-Zählers eip), z.B. jmp, call, ret.
- Basisblöcke werden nach sensiblen Instruktionen abgesucht
- diese werden jeweils durchAufruf einer HV-Prozedur ersetzt, die jeweilige Instruktion behandelt
- gleiche Verfahrensweise mit letzter Instruktion eines Basis-Blocks

Mechanismus: Binary Translation (Binärcodeübersetzung)

- modifizierter Basisblock: wird innerhalbdes HVin Cachegespeichert und ausgeführt
- Basisblock ohne sensible Instruktionen: läuft unter Typ-2-HV exakt so schnell wie unmittelbar auf Hardware (weil er auch tatsächlich unmittelbar auf der Hardware läuft, nur eben im HV-Kontext)
- sensible Instruktionen: nach dargestellter Methode abgefangen und emuliert → dabei hilft jetzt das Host-BS (z.B. durch eigene Systemaufrufe, Gerätetreiberschnittstellen)

Mechanismus: Caching von Basisblöcken

- HV nutzt zwei parallel arbeitende Module (Host-BS-Threads!):
  - Translator: Code-Inspektion, Binary Translation
  - Dispatcher: Basisblock-Ausführung
- zusätzliche Datenstruktur: Basisblock-Cache
- Dispatcher: sucht Basisblock mit jeweils nächster auszuführender Befehlsadresse im Cache; falls miss → suspendieren (zugunsten Translator)
- Translator: schreibt Basisblöcke in Basisblock-Cache
- Annahme: irgendwann ist Großteil des Programms im Cache, dieses läuft dann mit nahezu Original-Geschwindigkeit (theoretisch)

Performanzmessungen

- zeigen gemischtes Bild: Typ2-HV keinesfalls so schlecht, wie einst erwartet wurde
- qualitativer Vergleich mit virtualisierbarer Hardware (Typ1-Hypervisor):
- ,,trap-and-emulate,,: erzeugt Vielzahl von Traps → Kontextwechsel zwischen jeweiliger VM und HV
- insbesondere bei Vielzahl an VMs sehr teuer: CPU-Caches, TLBs, Heuristiken zur spekulativen Ausführung werden verschmutzt
- wenn andererseits sensible Instruktionen durch Aufruf von VMware-Prozeduren innerhalb des ausführenden Programms ersetzt: keine Kontextwechsel-Overheads

Studie: (von Vmware) [Adams&Agesen06]

- last-und anwendungsabhängig kann Softwarelösung sogar Hardwarelösung übertreffen
- Folge: viele moderne Typ1-HV benutzen aus Performanzgründen ebenfalls Binary Translation

Paravirtualisierung

Funktionsprinzip

- ... unterscheidet sich prinzipiell von Typ-1/2-Hypervisor
- wesentlich: Quellcode des Gast-Betriebssystems modifiziert
- sensible Instruktionen: durch Hypervisor-Calls ersetzt
- Folge: Gast-Betriebssystem arbeitet jetzt vollständig wie Nutzerprogramm, welches Systemaufrufe zum Betriebssystem (hier dem Hypervisor) ausführt
- dazu:
  - Hypervisor: muss geeignetes Interface definieren (HV-Calls)
  - → Menge von Prozedur-Aufrufen zur Benutzung durch Gast-Betriebssystem
  - bilden eine HV-API als Schnittstelle für Gast-Betriebssysteme (nicht für Nutzerprogramme!)
- mehr dazu: Xen

Verwandtschaft mit Mikrokernel-Architekturen

- Geht man vom Typ-1-HV noch einen Schritt weiter ...
  - und entfernt alle sensiblen Instruktionen aus Gast-Betriebssystem ...
  - und ersetzt diese durch Hypervisor-Aufrufe, um Systemdienste wie E/A zu benutzen, ...
  - hat man praktisch den Hypervisor in Mikrokernel transformiert.
- ... und genau das wird auch schon gemacht: L^4Linux (TU Dresden)
  - Basis: stringente $L^4\mu$ Kernel-Implementierung (Typ-1-HV-artiger Funktionsumfang)
  - Anwendungslaufzeitumgebung: paravirtualisierter Linux-Kernel als Serverprozess
  - Ziele: Isolation (Sicherheit, Robustheit), Echtzeitfähigkeit durch direktere HW-Interaktion (vergleichbar Exokernel-Ziel)

Zwischenfazit Virtualisierung

- Ziele: Adaptivität komplementär zu...
  - Wartbarkeit : ökonomischer Betrieb von Cloud-und Legacy-Anwendungen ohne dedizierte Hardware
  - Sicherheit : sicherheitskritische Anwendungen können vollständig von nichtvertrauenswürdigen Anwendungen (und untereinander) isoliert werden
  - Robustheit : Fehler in VMs (= Anwendungsdomänen) können nicht andere VMs beeinträchtigen
- Idee: drei gängige Prinzipien:
  - Typ-1-HV: unmittelbares HW-Multiplexing, trap-and-emulate
  - Typ-2-HV: HW-Multiplexing auf Basis eines Host-OS, binarytranslation
  - Paravirtualisierung: Typ-1-HV für angepasstes Gast-OS, kein trap-and-emulate nötig → HV ähnelt μKern
- Ergebnisse:
  - ✓ VMs mit individuell anpassbarer Laufzeitumgebung
  - ✓ isolierteVMs
  - ✓ kontrollierbare VM-Interaktion (untereinander und mit HW)
  - ✗ keine hardwarespezifischen Optimierungen aus VM heraus möglich → Performanz, Echtzeitfähigkeit, Sparsamkeit!

Container

Ziele:

- Adaptivität , im Dienste von ...
- ... Wartbarkeit: einfachen Entwicklung, Installation, Rekonfiguration durch Kapselung von
  - Anwendungsprogrammen
  - * durch sie benutzte Bibliotheken
  - * Instanzen bestimmter BS-Ressourcen
- ... Portabilität: Betrieb von Anwendungen, die lediglich von einem bestimmten BS-Kernel abhängig sind (nämlich ein solcher, der Container unterstützt); insbesondere hinsichtlich:
  - Abhängigkeitskonflikten (Anwendungen und Bibliotheken)
  - fehlenden Abhängigkeiten (Anwendungen und Bibliotheken)
  - Versions-und Namenskonflikten
- ... Sparsamkeit: problemgerechtes ,,Packen,, von Anwendungen in Container → Reduktion an Overhead: selten (oder gar nicht) genutzter Code, Speicherbedarf, Hardware, ...

Idee:

- private Sichten (Container) bilden = private User-Space-Instanzen für verschiedene Anwendungsprogramme
- Kontrolle dieser Container i.S.v. Multiplexing, Unabhängigkeit und API: BS-Kernel
- somit keine Form der BS-Virtualisierung, eher: ,,User-Space-Virtualisierung,,

Anwendungsfälle für Container

- Anwendungsentwicklung:
  - konfliktfreies Entwickeln und Testen unterschiedlicher Software, für unterschiedliche Zielkonfigurationen BS-User-Space
- Anwendungsbetrieb und -administration:
  - Entschärfung von ,,dependency hell,,

  - einfache Migration, einfaches Backup von Anwendungen ohne den (bei Virtualisierungsimages als Ballast auftretenden) BS-Kernel
  - einfache Verteilung generischer Container für bestimmte Aufgaben
  - = Kombinationen von Anwendungen
- Anwendungsisolation? → Docker

Zwischenfazit: Container

- Ziele: Adaptivität komplementär zu...
  - Wartbarkeit : Vermeidung von Administrationskosten für Laufzeitumgebung von Anwendungen
  - Portabilität : Vereinfachung von Abhängigkeitsverwaltung
  - Sparsamkeit : Optimierung der Speicher-und Verwaltungskosten für Laufzeitumgebung von Anwendungen
- Idee:
  - unabhängige User-Space-Instanz für jeden einzelnen Container
  - Aufgaben des Kernels: Unterstützung der Containersoftware bei Multiplexing und Herstellung der Unabhängigkeitdieser Instanzen
- Ergebnisse:
  - ✓ vereinfachte Anwendungsentwicklung
  - ✓ vereinfachter Anwendungsbetrieb
  - ✗ Infrastruktur nötig über (lokale) Containersoftware hinaus, um Containern zweckgerecht bereitzustellen und zu warten
  - ✗ keine vollständige Isolationmöglich

Beispielsysteme (Auswahl)

- Virtualisierung: VMware, VirtualBox
- Paravirtualisierung: Xen
- Exokernel: Nemesis, MirageOS, RustyHermit
- Container: Docker, LupineLinux

Hypervisor

VMware

- "... ist Unternehmenin PaloAlto, Kalifornien (USA)
- gegründet 1998 von 5 Informatikern
- stellt verschiedene Virtualisierungs-Softwareprodukte her:
  1. VMware Workstation
     - war erstes Produkt von VMware (1999)
     - mehrere unabhängige Instanzen von x86- bzw. x86-64-Betriebssystemen auf einer Hardware betreibbar
  2. VMware Fusion: ähnliches Produkt für Intel Mac-Plattformen
  3. VMware Player: (eingestellte) Freeware für nichtkommerziellen Gebrauch
  4. VMware Server (eingestellte Freeware, ehem. GSX Server)
  5. VMware vSphere (ESXi)
     - Produkte 1 ... 3: für Desktop-Systeme
     - Produkte 4 ... 5: für Server-Systeme
     - Produkte 1 ... 4: Typ-2-Hypervisor
- bei VMware-Installation: spezielle vm- Treiber in Host-Betriebssystem eingefügt
- diese ermöglichen: direkten Hardware-Zugriff
- durch Laden der Treiber: entsteht ,,Virtualisierungsschicht" (VMware-Sprechweise)
  - Typ1- Hypervisor- Architektur
  - Anwendung nur bei VMware ESXi
  - Entsprechende Produkte in Vorbereitung

VirtualBox

- Virtualisierungs-Software für x86- bzw. x86-64-Betriebssysteme für Industrie und ,,Hausgebrauch" (ursprünglich: Innotek , dann Sun , jetzt Oracle)
- frei verfügbare professionelle Lösung, als Open Source Software unter GNU General Public License(GPL) version 2. ...
- (gegenwärtig) lauffähig auf Windows, Linux, Macintosh und Solaris Hosts
- unterstützt große Anzahl von Gast-Betriebssystemen: Windows (NT 4.0, 2000, XP, Server 2003, Vista, Windows 7), DOS/Windows 3.x, Linux (2.4 and 2.6), Solaris and OpenSolaris , OS/2 , and OpenBSD u.a.
- reiner Typ-2-Hypervisor

Paravirutalisierung: Xen

- entstanden als Forschungsprojekt der University of Cambridge (UK), dann XenSource Inc., danach Citrix, jetzt: Linux Foundation (,,self-governing")
- frei verfügbar als Open Source Software unter GNU General Public License (GPL)
- lauffähig auf Prozessoren der Typen x86, x86-64, PowerPC, ARM, MIPS
- unterstützt große Anzahl von Gast-Betriebssystemen: FreeBSD, GNU/Hurd/Mach, Linux, MINIX, NetBSD, Netware, OpenSolaris, OZONE, Plan 9
- ,,Built for the cloud before it was called cloud." (Russel Pavlicek, Citrix)
- bekannt für Paravirtualisierung
- unterstützt heute auch andere Virtualisierungs-Prinzipien

Xen : Architektur

- Gast-BSe laufen in Xen Domänen (,,dom_i", analog VM_i)
- es existiert genau eine, obligatorische, vertrauenswürdige Domäne: dom_0
- Aufgaben (Details umseitig):
  - Bereitstellen und Verwalten der virtualisierten Hardware für andere Domänen (Hypervisor-API, Scheduling-Politiken für Hardware-Multiplexing)
  - Hardwareverwaltung/-kommunikation für paravirtualisierte Gast-BSe (Gerätetreiber)
  - Interaktionskontrolle (Sicherheitspolitiken)
- dom_0 im Detail: ein separates, hochkritisch administriertes, vertrauenswürdiges BS mit eben solchen Anwendungen (bzw. Kernelmodulen) zur Verwaltung des gesamten virtualisierten Systems
  - es existieren hierfür spezialisierte Variantenvon Linux, BSD, GNU Hurd

Xen : Sicherheit

- Sicherheitsmechanismusin Xen: Xen Security Modules (XSM)
- illustriert, wie (Para-) Typ-1-Virtualisierung von BS die NFE Sicherheit unterstützt
- PDP: Teil des vertrauenswürdigen BS in dom_0, PEPs: XSMs im Hypervisor
- Beispiel: Zugriff auf Hardware
  - Sicherheitspolitik-Integration, Administration, Auswertung: dom_0
- Beispiel: Inter-Domänen-Kommunikation
  - Interaktionskontrolle (Aufgaben wie oben): dom_0
  - Beispiel: VisorFlow
  - selber XSM kontrolliert Kommunikation für zwei Domänen

Exokernel

Nemesis

- Betriebssystemaus EU-Verbundprojekt ,,Pegasus,, zur Realisierung eines verteilten multimediafähigen Systems (1. Version: 1994/95)
- Entwurfsprinzipien:
  1. Anwendungen: sollen Freiheit haben, Betriebsmittel in für sie geeignetster Weise zu nutzen (= Exokernel-Prinzip)
  2. Realisierung als sog. vertikal strukturiertes Betriebssystem:
     - weitaus meiste Betriebssystem-Funktionalität innerhalb der Anwendungen ausgeführt (= Exokernel-Prinzip)
     - Echtzeitanforderungen durch Multimedia → Vermeidung von Client-Server-Kommunikationsmodell wegen schlecht beherrschbarer zeitlicher Verzögerungen (neu)

MirageOS + Xen

- Spezialfall: Exokernel als paravirtualisiertes BS auf Xen
- Ziele : Wartbarkeit (Herkunft: Virtualisierungsarchitekturen ...)
  - ökonomischer HW-Einsatz
  - Unterstützung einfacher Anwendungsentwicklung
  - nicht explizit: Unterstützung von Legacy-Anwendungen!
- Idee: ,,Unikernel" → eine Anwendung, eine API, ein Kernel
- umfangreiche Dokumentation, Tutorials, ... → ausprobieren
- Unikernel - Idee
  - Architekturprinzip:
  - in MirageOS:
- Ergebnis: Kombination von Vorteilen zweier Welten
  - Virtualisierungs vorteile: Sicherheit, Robustheit (→ Xen - Prinzip genau einer vertrauenswürdigen, isolierten Domäne dom_0)
  - Exokernelvorteile: Wartbarkeit, Sparsamkeit
  - nicht: Exokernelvorteil der hardwarenahen Anwendungsentwicklung... (→ Performanz und Echzeitfähigkeit)

Container: Docker

- Idee: Container für einfache Wartbarkeit von Linux-Anwendungsprogrammen ...
  - ... entwickeln
  - ... testen
  - ... konfigurieren
  - ... portieren → Portabilität
- Besonderheit: Container können - unabhängig von ihrem Einsatzzweck - wie Software-Repositories benutzt, verwaltet, aktualisiert, verteilt ... werden
- Management von Containers: Docker Client → leichtgewichtiger Ansatz zur Nutzung der Wartbarkeitsvorteile von Virtualisierung
- Forsetzung unter der OCI (Open Container Initiative)
  - ,,Docker does a nice job [...] for a focused purpose, namely the lightweight packaging and deployment of applications." (Dirk Merkel, Linux Journal)
- Implementierung der Containertechnik basierend auf Linux-Kernelfunktionen:
  - Linux Containers (LXC): BS-Unterstützung für Containermanagement
  - cgroups: Accounting/Beschränkung der Ressourcenzuordnung
  - union mounting: Funktion zur logischen Reorganisation hierarchischer Dateisysteme

Performanz und Parallelität

Motivation

- Performanz: Wer hätte gern einen schnell(er)en Rechner...?
- Wer braucht schnelle Rechner:
  - Hochleistungsrechnen, HPC (,,high performancecomputing")
    - wissenschaftliches Rechnen(z.B. Modellsimulation natürlicher Prozesse, Radioteleskop-Datenverarbeitung)
    - Datenvisualisierung(z.B. Analysen großer Netzwerke)
    - Datenorganisation-und speicherung(z.B. Kundendatenverarbeitung zur Personalisierung von Werbeaktivitäten, Bürgerdatenverarbeitung zur Personalisierung von Geheimdienstaktivitäten)
  - nicht disjunkt dazu: kommerzielle Anwendungen
    - ,,Big Data": Dienstleistungen für Kunden, die o. g. Probleme auf gigantischen Eingabedatenmengen zu lösen haben (Software wie Apache Hadoop)
    - Wettervorhersage
  - anspruchsvolle Multimedia- Anwendungen
    - Animationsfilme
    - VR-Rendering

Performanzbegriff

- Performance: The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage. (IEEE)
- Performanz im engeren Sinne dieses Kapitels: Minimierung der für korrekte Funktion (= Lösung eines Berechnungsproblems) zur Verfügung stehenden Zeit.
- oder technischer: Maximierung der Anzahl pro Zeiteinheit abgeschlossener Berechnungen.

Roadmap

- Grundlegende Erkenntnis: Performanz geht nicht (mehr) ohne Parallelität → Hochleistungsrechnen = hochparalleles Rechnen
- daher in diesem Kapitel: Anforderungen hochparallelen Rechnens an ...
  - Hardware: Prozessorarchitekturen
  - Systemsoftware: Betriebssystemmechanismen
  - Anwendungssoftware: Parallelisierbarkeitvon Problemen
- BS-Architekturen anhand von Beispielsystemen:
  - Multikernel: Barrelfish
  - verteilte Betriebssysteme

Hardware-Voraussetzungen

- Entwicklungstendenzen der Rechnerhardware:
  - Multicore-Prozessoren: seit ca. 2006 (in größerem Umfang)
  - Warum neues Paradigma für Prozessoren? bei CPU-Taktfrequenz >> 4 GHz: z.Zt. physikalische Grenze, u.a. nicht mehr sinnvoll handhabbare Abwärme
  - Damit weiterhin:
    1. Anzahl der Kerne wächst nicht linear
    2. Taktfrequenz wächst asymptotisch, nimmt nur noch marginal zu

Performanz durch Parallelisierung ...

Folgerungen

1. weitere Performanz-Steigerung von Anwendungen: primär durch Parallelität (aggressiverer) Multi-Threaded-Anwendungen
2. erforderlich: Betriebssystem-Unterstützung → Scheduling, Sychronisation
3. weiterhin erforderlich: Formulierungsmöglichkeiten (Sprachen), Compiler, verteilte Algorithmen ... → hier nicht im Fokus

... auf Prozessorebene

Vorteile von Multicore-Prozessoren

1. möglich wird: **Parallelarbeit auf Chip-Ebene** → Vermeidung der Plagen paralleler verteilter Systeme
2. bei geeigneter Architektur: Erkenntnisse und Software aus Gebiet verteilter Systeme als Grundlage verwendbar
3. durch gemeinsame Caches (architekturabhängig): schnellere Kommunikation (speicherbasiert), billigere Migration von Aktivitäten kann möglich sein
4. höhere Energieeffizienz: mehr Rechenleistung pro Chipfläche, geringere elektrische Leistungsaufnahme → weniger Gesamtabwärme, z.T. einzelne Kerne abschaltbar (vgl. Sparsamkeit , mobile Geräte)
5. Baugröße: geringeres physisches Volumen

Nachteile von Multicore-Prozessoren

1. durch gemeinsam genutzte Caches und Busstrukturen: Engpässe (Bottlenecks) möglich
2. zur Vermeidung thermischer Zerstörungen: Lastausgleich zwingend erforderlich! (Ziel: ausgeglichene Lastverteilung auf einzelnen Kernen)
3. zum optimalen Einsatz zwingend erforderlich:
   (a) Entwicklung Hardwarearchitektur
   (b) zusätzlich: Entwicklung geeigneter Systemsoftware
   (c) zusätzlich: Entwicklung geeigneter Anwendungssoftware

Multicore-Prozessoren

- Sprechweise in der Literatur gelegentlich unübersichtlich...
- daher: Terminologie und Abkürzungen:
  - MC ...multicore(processor)
  - CMP ...chip-level multiprocessing, hochintegrierte Bauweise für ,,MC"
  - SMC ...symmetric multicore → SMP ... symmetric multi-processing
  - AMC ...asymmetric (auch: heterogeneous) multicore → AMP ... asymmetric multi-processing
  - UP ...uni-processing , Synonym zu singlecore(SC) oder uniprocessor

Architekturen von Multicore-Prozessoren

- A. Netzwerkbasiertes Design
  - Prozessorkerne des Chips u. ihre lokalen Speicher (oder Caches): durch Netzwerkstruktur verbunden
  - damit: größte Ähnlichkeit zu traditionellen verteilten Systemen
  - Verwendung: bei Vielzahl von Prozessorkernen (Skalierbarkeit!)
  - Beispiel: Intel Teraflop-Forschungsprozessor Polaris (80 Kerne als 8x10-Gitter)
- B. Hierarchisches Design
  - mehrere Prozessor-Kerne teilen sich mehrere baumartig angeordnete Caches
  - meistens:
    - jeder Prozessorkern hat eigenen L1-Cache
    - L2-Cache, Zugriff auf (externen) Hauptspeicher u. Großteil der Busse aber geteilt
  - Verwendung: typischerweise Serverkonfigurationen
  - Beispiele:
    - IBM Power
    - Intel Core 2, Core i
    - Sun UltraSPARCT1 (Niagara)
- C. Pipeline-Design
  - Daten durch mehrere Prozessor-Kerne schrittweise verarbeitet
  - durch letzten Prozessor: Ablage im Speichersystem
  - Verwendung:
    - Graphikchips
    - (hochspezialisierte) Netzwerkprozessoren
  - Beispiele: Prozessoren X10 u. X11 von Xelerator zur Verarbeitung von Netzwerkpaketen in Hochleistungsroutern (X11: bis zu 800 Pipeline-Prozessorkerne)

Symmetrische u. asymmetrische Multicore-Prozessoren

- symmetrische Multicore-Prozessoren (SMC)
  - alle Kerne identisch, d.h. gleiche Architektur und gleiche Fähigkeiten
  - Beispiele:
    - Intel Core 2 Duo
    - Intel Core 2 Quad
    - ParallaxPropeller
- asymmetrische MC-Prozessoren (AMC)
- Multicore-Architektur, jedoch mit Kernen unterschiedlicher Architektur und/oder unterschiedlichen Fähigkeiten
- Beispiel: Kilocore:
  - 1 Allzweck-Prozessor (PowerPC)
  - 256 od. 1024 Datenverarbeitungsprozessoren

Superskalare Prozessoren

- Bekannt aus Rechnerarchitektur: Pipelining
  - parallele Abarbeitung von Teilen eines Maschinenbefehls in Pipeline-Stufen
  - ermöglicht durch verschiedene Funktionseinheiten eines Prozessors für verschiedene Stufen:
    - Control Unit (CU)
    - ArithmeticLogicUnit (ALU)
    - Float Point Unit (FPU)
    - Memory Management Unit (MMU)
    - Cache
  - sowie mehrere Pipeline-Register
- superskalare Prozessoren: solche, bei denen zur Bearbeitung einer Pipeling-Stufe erforderlichen Funktionseinheiten n-fach vorliegen
- Ziel:
  - Skalarprozessor (mit Pipelining): 1 Befehl pro Takt (vollständig) bearbeitet
  - Superskalarprozessor: bis zu n Befehle pro Taktbearbeitet
- Verbereitung heute: universell (bis hin zu allen Desktop-Prozessorfamilien)

Parallelisierung in Betriebssystemen

- Basis für alle Parallelarbeit aus BS-Sicht: Multithreading
- wir erinnern uns ...:
  - Kernel-Level-Threads (KLTs): BS implementiert Threads → Scheduler kann mehrere Threads nebenläufig planen → Parallelität möglich
  - User-Level-Threads (ULTs): Anwendung implementiert Threads → keine Parallelität möglich!
- grundlegend für echt paralleles Multithreading:
  - parallelisierungsfähige Hardware
  - kausal unabhängige Threads
  - passendes (und korrekt eingesetztes!) Programmiermodell, insbesondere Synchronisation!

  → Programmierer + Compiler

Vorläufiges Fazit:

- BS-Abstraktionen müssen Parallelität unterstützen (Abstraktion nebenläufiger Aktivitäten: KLTs)
- BS muss Synchronisationsmechanismen implementieren

Synchronisations- und Sperrmechanismen

- Synchronisationsmechanismen zur Nutzung
  - ... durch Anwendungen → Teil der API
  - ... durch den Kernel (z.B. Implementierung Prozessmanagement, E/A, ...)
- Aufgabe: Verhinderung konkurrierender Zugriffe auf logische oder physische Ressourcen
  - Vermeidung von raceconditions
  - Herstellung einer korrekten Ordnung entsprechend Kommunikationssemantik (z.B. ,,Schreiben vor Lesen")
- (alt-) bekanntes Bsp.: Reader-Writer-Problem

Erinnerung: Reader-Writer-Problem

- Begriffe: (bekannt)
  - wechselseitiger Ausschluss (mutual exclusion)
  - kritischer Abschnitt (critical section)
- Synchronisationsprobleme:
  - Wie verhindern wir ein write in vollen Puffer?
  - Wie verhindern wir ein read aus leerem Puffer?
  - Wie verhindern wir, dass auf ein Element während des read durch ein gleichzeitiges write zugegriffen wird? (Oder umgekehrt?)

Sperrmechanismen (Locks)

- Wechselseitiger Ausschluss ...
  - ... ist in nebenläufigen Systemen zwingend erforderlich
  - ... ist in echt parallelen Systemen allgegenwärtig
  - ... skaliert äußerst unfreundlich mit Code-Komplexität → (monolithischer) Kernel-Code!
- Mechanismen in Betriebssystemen: Locks
- Arten von Locks am Beispiel Linux:
  - Big Kernel Lock (BKL)
    - historisch (1996-2011): lockkernel(); ... unlockkernel();
    - ineffizient durch massiv gestiegene Komplexität des Kernels
  - atomic-Operationen
  - Spinlocks
  - Semaphore (Spezialform: Reader/Writer Locks)

atomic*

- Bausteine der komplexeren Sperrmechanismen:
  - Granularität: einzelne Integer- (oder sogar Bit-) Operation
  - Performanz: mittels Assembler implementiert, nutzt Atomaritäts garantiender CPU (TSL - Anweisungen: ,,test-set-lock")
- Benutzung:
  - `atomic_*` Geschmacksrichtungen: read, set, add, sub, inc, dec u. a.
  - keine explizite Lock-Datenstruktur → Deadlocks durch Mehrfachsperrung syntaktisch unmöglich
  - definierte Länge des kritischen Abschnitts (genau diese eine Operation) → unnötiges Sperren sehr preiswert

Zusammenfassung

Funktionale und nichtfunktionale Eigenschaften

- Funktionale Eigenschaften: beschreiben, was ein (Software)-Produkt tun soll
- Nichtfunktionale Eigenschaften: beschreiben, wie funktionale Eigenschaften realisiert werden, also welche sonstigen Eigenschaftendas Produkt haben soll ... unterteilbar in:
  1. Laufzeiteigenschaften (zur Laufzeit sichtbar)
  2. Evolutionseigenschaften (beim Betrieb sichtbar: Erweiterung, Wartung, Test usw.)

Roadmap (... von Betriebssystemen)

- Sparsamkeit und Effizienz
- Robustheit und Verfügbarkeit
- Sicherheit
- Echtzeitfähigkeit
- Adaptivität
- Performanzund Parallelität

Sparsamkeit und Effizienz

- Sparsamkeit: Die Eigenschaft eines Systems, seine Funktion mit minimalem Ressourcenverbrauch auszuüben.
- Effizienz: Der Grad, zu welchem ein System oder eine seiner Komponenten seine Funktion mit minimalem Ressourcenverbrauch ausübt. → Ausnutzungsgrad begrenzter Ressourcen
- Die jeweils betrachtete(n) Ressource(n) muss /(müssen) dabei spezifiziert sein!
- sinnvolle Möglichkeiten bei Betriebssystemen:
  1. Sparsamer Umgang mit Energie , z.B. energieeffizientes Scheduling
  2. Sparsamer Umgang mit Speicherplatz (Speichereffizienz)
  3. Sparsamer Umgang mit Prozessorzeit
  4. ...

Sparsamkeit mit Energie

- Sparsamkeit mit Energie als heute extrem wichtigen Ressource, mit nochmals gesteigerter Bedeutung bei mobilen bzw. vollständig autonomen Geräten Maßnahmen:

1. Hardware-Ebene: momentan nicht oder nicht mit maximaler Leistung benötigte Ressourcen in energiesparenden Modus bringen: abschalten, Standby, Betrieb mit verringertem Energieverbrauch (abwägen gegen verminderte Leistung). (Geeignete Hardware wurde/wird ggf. erst entwickelt)
2. Software-Ebene: neue Komponenten entwickeln, die in der Lage sein müssen:
   - Bedingungenzu erkennen, unter denen ein energiesparender Modus möglich ist;
   - Steuerungs-Algorithmen für Hardwarebetrieb so zu gestalten, dass Hardware-Ressourcen möglichst lange in einem energiesparenden Modus betrieben werden.
   - Energie-Verwaltungsstrategien: energieeffizientes Scheduling zur Vermeidung von Unfairness und Prioritätsumkehr
   - Beispiele: energieeffizientes Magnetfestplatten-Prefetching, energiebewusstes RR-Scheduling

Sparsamkeit mit Speicherplatz

- Betrachtet: Sparsamkeit mit Speicherplatz mit besonderer Wichtigkeit für physisch beschränkte, eingebettete und autonome Geräte
- Maßnahmen Hauptspeicherauslastung:
  1. Algorithmus und Strategie z.B.:
     - Speicherplatz sparende Algorithmen zur Realisierung gleicher Strategien
  2. Speicherverwaltung von Betriebssystemen:
     - physische vs. virtuelle Speicherverwaltung
     - speichereffiziente Ressourcenverwaltung
     - Speicherbedarfdes Kernels
     - direkte Speicherverwaltungskosten
- Maßnahmen Hintergrundspeicherauslastung:
  1. Speicherbedarf des Betriebssystem-Images
  2. dynamische SharedLibraries
  3. VMM-Auslagerungsbereich
  4. Modularität und Adaptivität des Betriebssystem-Images
- Nicht betrachtet: Sparsamkeit mit Prozessorzeit → 99% Überschneidung mit NFE Performanz

Robustheit und Verfügbarkeit

- Robustheit: Zuverlässigkeit unter Anwesenheit externer Ausfälle
- fault, aktiviert → error, breitet sich aus → failure

Robustheit

- Erhöhung der Robustheit durch Isolation:
  - Maßnahmen zur Verhinderung der Fehlerausbreitung:
  1. Adressraumisolation: Mikrokernarchitekturen,
  2. kryptografische HW-Unterstützung: Intel SGX und
  3. Virtualisierungsarchitekturen
- Erhöhung der Robustheit durch Behandlung von Ausfällen: Micro-Reboots

Vorbedingung für Robustheit: Korrektheit

- Korrektheit: Eigenschaft eines Systems sich gemäß seiner Spezifikation zu verhalten (unter der Annahme, dass bei dieser keine Fehler gemacht wurden).
- Maßnahmen (nur angesprochen):

1. diverse Software-Tests:
   - können nur Fehler aufspüren, aber keine Fehlerfreiheit garantieren!
2. Verifizierung:
   - Durch umfangreichen mathematischen Apparat wird Korrektheit der Software bewiesen.
   - Aufgrund der Komplexität ist Größe verifizierbarer Systeme (bisher?) begrenzt.
   - Betriebssystem-Beispiel: verifizierter Mikrokern seL

Verfügbarkeit

- Verfügbarkeit: Der Anteil an Laufzeit eines Systems, in dem dieses seine spezifizierte Leistung erbringt.
- angesprochen: Hochverfügbare Systeme
- Maßnahmen zur Erhöhung der Verfügbarkeit:
  1. Robustheitsmaßnahmen
  2. Redundanz
  3. Redundanz
  4. Redundanz
  5. Ausfallmanagement

Sicherheit

- Sicherheit (IT-Security): Schutz eines Systems gegen Schäden durch zielgerichtete Angriffe, insbesondere in Bezug auf die Informationen, die es speichert, verarbeitet und kommuniziert.
- Sicherheitsziele:
  1. Vertraulichkeit (Confidentiality)
  2. Integrität (Integrity)
  3. Verfügbarkeit (Availability)
  4. Authentizität (Authenticity)
  5. Verbindlichkeit (Non-repudiability)

Security Engineering

- Sicherheitsziele → Sicherheitspolitik → Sicherheitsarchitektur → Sicherheitsmechanismen
- Sicherheitspolitik: Regeln zum Erreichen eines Sicherheitsziels.

  - hierzu formale Sicherheitsmodelle:
  - IBAC, TE, MLS
  - DAC, MAC
- Sicherheitsmechanismen: Implementierung der Durchsetzung einer Sicherheitspolitik.
  - Zugriffssteuerungslisten(ACLs)
  - SELinux
- Sicherheitsarchitektur: Platzierung, Struktur und Interaktion von Sicherheitsmechanismen.
  - wesentlich: Referenzmonitorprinzipien
  - RM1: Unumgehbarkeit → vollständiges Finden aller Schnittstellen
  - RM2: Manipulationssicherheit → Sicherheit einerSicherheitspolitik selbst
  - RM3: Verifizierbarkeit → wohlstrukturierte und per Designkleine TCBs

Echtzeitfähigkeit

- Echtzeitfähigkeit: Fähigkeit eines Systems auf eine Eingabe innerhalb eines spezifizierten Zeitintervalls eine korrekte Reaktion hervorzubringen.
- Maximum dieses relativen Zeitintervalls: Frist d

1. echtzeitfähige Scheduling-Algorithmen für Prozessoren
   - zentral: garantierte Einhaltung von Fristen
   - wichtige Probleme: Prioritätsumkehr, Überlast, kausale Abhängigkeit
2. echtzeitfähige Interrupt-Behandlung
   - zweiteilig:asynchron registrieren, geplant bearbeiten
3. echtzeitfähige Speicherverwaltung
   - Primärspeicherverwaltung, VMM (Pinning)
   - Sekundärspeicherverwaltung, Festplattenscheduling

Adaptivität

- Adaptivität: Eigenschaft eines Systems, so gebaut zu sein, dass es ein gegebenes (breites) Spektrum nichtfunktionaler Eigenschaften unterstützt.
- Beobachtung: Adaptivität i.d.R. als komplementär und synergetisch zu anderen NFE:
  - Sparsamkeit
  - Robustheit
  - Sicherheit
  - Echzeitfähigkeit
  - Performanz
  - Wartbarkeit und Portierbarkeit

Adaptive Systemarchitekturen

- Zielstellungen:
  - Exokernel: { Adaptivität } ∪ { Performanz, Echtzeitfähigkeit, Wartbarkeit, Sparsamkeit }
  - Virtualisierung: { Adaptivität } ∪ { Wartbarkeit, Sicherheit, Robustheit }
  - Container: { Adaptivität } ∪ { Wartbarkeit, Portabilität, Sparsamkeit }

Performanz und Parallelität

- Performanz (wie hier besprochen): Eigenschaft eines Systems, die für korrekte Funktion (= Berechnung) benötigte Zeit zu minimieren.
- hier betrachtet: Kurze Antwort-und Reaktionszeiten
  1. vor allen Dingen: Parallelisierung auf Betriebssystemebene zur weiteren Steigerung der Performanz/Ausnutzung von Multicore-Prozessoren(da Steigerung der Prozessortaktfrequenz kaum noch möglich)
  2. weiterhin: Parallelisierung auf Anwendungsebene zur Verringerung der Antwortzeiten von Anwendungen und Grenzen der Parallelisierbarkeit(für Anwendungen auf einem Multicore-Betriebssystem).

Mechanismen, Architekturen, Grenzen der Parallelisierung

- Hardware:
  - Multicore-Prozessoren
  - Superskalarität
- Betriebssystem:
  - Multithreading(KLTs) und Scheduling
  - Synchronisation und Kommunikation
  - Lastangleichung
- Anwendung(sprogrammierer):
  - Parallelisierbarkeiteines Problems
  - optimaler Prozessoreneinsatz, Effizienz

Synergetische und konträre Eigenschaften

- Normalerweise:
  - Eine nichtfunktionale Eigenschaft bei IT-Systemen meist nicht ausreichend
  - Beispiel: Was nützt ein Echtzeit-Betriebssystem - z.B. innerhalb einer Flugzeugsteuerung - wenn es nicht auch verlässlich arbeitet?
- In diesem Zusammenhang interessant:
  - Welche nichtfunktionalen Eigenschaften mit Maßnahmen erreichbar, die in gleiche Richtung zielen, bei welchen wirken Maßnahmen eher gegenläufig?
  - Erstere sollen synergetische, die zweiten konträre (also in Widerspruch zueinander stehende) nichtfunktionale Eigenschaften genannt werden.
  - Zusammenhang nicht immer eindeutig und offensichtlich, wie z.B. bei: ,,Sicherheit kostet Zeit." (d.h. Performanz und Sicherheit sind nichtsynergetische Eigenschaften)

Notwendige NFE-Paarungen

- Motivation: Anwendungen (damit auch Betriebssysteme) für bestimmte Einsatzgebiete brauchen oft mehrere nichtfunktionale Eigenschaften gleichzeitig - unabhängig davon, ob sich diese synergetisch oder nichtsynergetisch zueinander verhalten.
- Beispiele:
  - Echtzeit und Verlässlichkeit: ,,SRÜ"-Systeme an potentiell gefährlichen Einsatzgebieten (Atomkraftwerk, Flugzeugsteuerung, Hinderniserkennung an Fahrzeugen, ...)
  - Echtzeit und Sparsamkeit: Teil der eingebetteten Systeme
  - Robustheit und Sparsamkeit: unter entsprechenden Umweltbedingungen eingesetzte autonome Systeme, z.B. smart-dust-Systeme

Überblick: NFE und Architekturkonzepte

- ✓ ... Zieleigenschaft
- (✓) ... synergetische Eigenschaft
- ✗ ... konträre Eigenschaft
- Leere Zellen: keine pauschale Aussage möglich.

Fazit: Breites und offenes Forschungsfeld → werden Sie aktiv!